B-Toolkit

User's Manual

Release 3.4

©Copyright B-Core(UK) Ltd. 1997

IMPORTANT NOTICE: This Document and B-Toolkit software are proprietary to B-Core(UK) Ltd.. They are supplied under licence from B-Core(UK) Ltd. and are not to be reproduced in whole or in part in any form nor used for any purpose except as specified in the terms and conditions of the Licence granted herewith or as agreed in writing by B-Core(UK) Ltd..

B-Core(UK) Ltd. gives no warranty that the B-Toolkit Software is error free and B-Core(UK) Ltd. shall not be liable for any indirect or consequential loss however caused which arises from any reliance placed on the use of information contained in this Document or the use of B-Toolkit Software.

©1996 *B-Core (UK) Ltd.*

B-Toolkit User's Manual

Overview

The **B-Toolkit** comprises a suite of software tools designed to support the rigorous or formal development of software systems using the **B-Method** ([7], [2]). The B-Method uses the notion of **Abstract Machines** to specify and design software systems. Abstract Machines are specified in terms of the Abstract Machine Notation (**AMN**) which is in turn based on the mathematical theory of **Generalised Substitutions**.

If AMN descriptions are considered as the source of a B development, then the products of the development process are

- executable code,
- design documentation, and
- listings of proofs for verification conditions.

The B-Toolkit provides a development environment automating the management of all associated files, ensuring that the entire development, including executable code, design documentation and proof listings, is always in a consistent state.

Currently the toolkit provides support for the following aspects of software development:

- **configuration management** through integrity and dependency management of specifications, designs, code and documentation.
- **specification and design analysis**, through syntax checkers, type checkers and a specification animator.
- **verification**, through a proof-obligation generator and automatic and interactive provers.
- **coding and code reuse**, through language translators and linkers, rapid prototyping facilities and a reusable specification/code module library.
- **design documentation**, through automatic production of fully cross-referenced and indexed type-set documents from source files.
- **evolution control**, through automatically re-checking and re-generating of specifications, designs, code and documentation after modifications to source files.

©1996 *B-Core (UK) Ltd.*

How this manual is organised

The bulk of this manual is derived directly from the on-line help available with the B-Toolkit. It is organised into ten sections:-

Section 1 Basic information about the B-Toolkit.

Section 2 Information about the B-Method and the Abstract Machine Notation.

Section 3 How to manage AMN constructs.

Section 4 How to use the B-Toolkit analysers: the Analyser (type checker) and Animator.

Section 5 How to carry out proof in the B-Toolkit.

Section 6 Using the B-Toolkit's construct generators.

Section 7 How to create and execute code.

Section 8 Creating documents from and AMN development.

Section 9 Information about the System and Team libraries in the B-Toolkit.

Section 10 Other utilities of the B-Toolkit.

Section 11 References and Index

The index can also be used to locate topics.

©1996 *B-Core (UK) Ltd.*

Contents

1 The B-Toolkit **1-3**

1.1 Getting Started 1-3

1.1.1 The Windows 1-3

1.1.2 The Mouse 1-5

1.1.3 Subsidiary Panels 1-5

1.1.4 The Top Bar 1-5

1.1.5 The Tool Bar 1-6

1.1.6 On-line Help 1-7

1.2 The B-Toolkit 1-8

1.2.1 Development Environment 1-9

1.2.2 The Motif Interface 1-10

1.2.3 Constructs Area 1-11

1.2.4 The Top Bar 1-11

1.2.5 The Information Panel 1-13

1.2.6 The Tool Bar 1-13

1.2.7 The Display Area 1-14

1.2.8 The B-Toolkit Environments 1-14

1.2.9 The Main Environment 1-15

1.2.10 The Provers Environment 1-15

1.2.11 The Generators Environment 1-15

1.2.12 The Translators Environment 1-16

1.2.13 The Documents Environment 1-16

1.3 Installation 1-17

1.3.1 Contents of the 5 Disks 1-17

1.3.2 Installation 1-18

1.3.3 Reading the Disks 1-18

 ©1996 *B-Core (UK) Ltd.*

1.3.4 Running the Install Script 1-18
1.3.5 Run-time Library Requirements 1-21
1.3.6 The B Licence File 1-21
1.3.7 Running the B-Toolkit 1-22
1.3.8 Errors 1-22
1.3.9 Increasing the Capacity of the B-Toolkit 1-23
1.4 The B-Toolkit Environment 1-24
1.4.1 Obligatory Environment Variables 1-24
1.4.2 Optional Environment Variables 1-24
1.4.3 Other Environmental Considerations 1-26
1.5 Options 1-27
1.5.1 Remake 1-27
1.5.2 Construct Display 1-27
1.5.3 Editor 1-27
1.5.4 Documents 1-28
1.5.5 Programming Language 1-28
1.5.6 DMU language 1-28
1.5.7 Interface 1-29

2 The B-Method 2-3

2.1 Introduction 2-3
2.2 The B-Method and the B-Toolkit 2-3
2.3 Abstract Machine Notation 2-5
2.3.1 The Generalised Substitution Language 2-5
2.4 Abstract Machine Specification 2-7
2.4.1 Structuring and Constructing Specifications 2-9
2.4.2 The INCLUDES, PROMOTES and EXTENDS Clauses 2-10
2.4.3 The USES and SEES Clauses 2-10

 ©1996 *B-Core (UK) Ltd.*

2.5 Abstract Machine Refinement 2-11
2.6 Abstract Machine Implementation 2-13
2.6.1 Structuring and Implementing Designs 2-14
2.6.2 The IMPORTS and SEES Clauses 2-14
2.7 The B-Toolkit Components 2-15
2.7.1 The B-Toolkit Managers 2-15
2.7.2 Analysis 2-15
2.7.3 Animation 2-16
2.7.4 Proof 2-16
2.7.5 Specification/Module Library 2-17
2.7.6 Code and Interface Generation 2-18
2.7.7 Enumerator and Base Generators 2-18
2.7.8 Documentation 2-18
2.8 Conclusion 2-19

3 Abstract Machine Notation (AMN) 3-3

3.1 Abstract Machine Notation 3-3
3.1.1 Overview 3-3
3.1.2 AMN Substitutions 3-3
3.1.3 AMN Operations 3-7
3.1.4 Abstract Machines 3-7
3.1.5 Machines 3-8
3.1.6 Refinements 3-12
3.1.7 Implementations 3-14
3.2 Mathematical Notation 3-18
3.2.1 Contents 3-18
3.2.2 Notes on operator binding 3-18
3.2.3 Infix Operator Priorities 3-19

 ©1996 *B-Core (UK) Ltd.*

3.2.4 Predicates . . . 3-19
3.2.5 General Predicates . . . 3-19
3.2.6 Predicates on Expressions . . . 3-20
3.2.7 Expressions . . . 3-20
3.2.8 Sets . . . 3-20
3.2.9 Set Expressions . . . 3-21
3.2.10 Natural Numbers . . . 3-22
3.2.11 Predicates on Natural Numbers . . . 3-23
3.2.12 Natural Number Expressions . . . 3-23
3.2.13 Relations . . . 3-24
3.2.14 Relational Expressions . . . 3-24
3.2.15 Functions . . . 3-26
3.2.16 Sequences . . . 3-27
3.2.17 Variables, Variable Lists and Identifiers . . . 3-28
3.2.18 Generalised Substitutions . . . 3-29
3.3 Structuring Developments . . . 3-30
3.3.1 Refinement . . . 3-30
3.3.2 Structuring Machines . . . 3-31
3.3.3 Purpose . . . 3-31
3.3.4 Including Machines . . . 3-31
3.3.5 Using Machines . . . 3-33
3.3.6 Non-Determinism in Included Machines . . . 3-33
3.3.7 Specification Structure and Proof . . . 3-33
3.3.8 Viewing the Structure of Specifications . . . 3-34
3.3.9 Structuring Designs . . . 3-34
3.3.10 Purpose . . . 3-34
3.3.11 Importing Machines . . . 3-34
3.3.12 Seeing Machines . . . 3-34

 ©1996 *B-Core (UK) Ltd.*

3.3.13 Hierarchical Design . . . 3-37
3.3.14 Design Structure and Proof . . . 3-39
3.3.15 Viewing the Structure of Designs . . . 3-39
3.4 ASCII Symbols for the Abstract Machine Notation . . . 3-40
3.4.1 Table of relation types . . . 3-40
3.4.2 AMN Symbols . . . 3-40
3.4.3 Generalised Substitution Language . . . 3-45

4 Construct Management 4-3
4.1 The Introduce Tool . . . 4-3
4.1.1 Introduce New Construct . . . 4-3
4.1.2 Introduce construct(s) from SRC . . . 4-4
4.1.3 Introduce construct from SLIB . . . 4-4
4.1.4 Introduce construct from TLIB . . . 4-4
4.2 The Editor . . . 4-6
4.2.1 Open Editor . . . 4-6
4.2.2 Quit Editor . . . 4-6
4.3 Close . . . 4-8
4.4 Commit . . . 4-9
4.5 ResetConstruct (rst) . . . 4-10
4.6 Rename . . . 4-11
4.7 Remove . . . 4-12
4.8 Status (sts) . . . 4-13
4.9 Hypertext View of a Development . . . 4-14
4.9.1 Referencing constructs . . . 4-14
4.9.2 Referenced constructs . . . 4-14
4.10 Development Overview . . . 4-16

5 Analysis 5-3

 ©1996 *B-Core (UK) Ltd.*

5.1 The Analyser (anl) 5-3
5.1.1 Edited Files 5-3
5.1.2 Subordinate Constructs 5-4
5.1.3 Type Determination 5-4
5.1.4 Type Checking 5-5
5.2 The Animator (anm) 5-6
5.2.1 Providing More Rewrite Rules 5-10
5.3 Mini-remake 5-12
5.4 Remake 5-13

6 Proof **6-3**

6.1 The B-Platform 6-3
6.1.1 Inference Rules 6-3
6.1.2 Simple Rules 6-4
6.1.3 Rewrite Rules 6-5
6.1.4 Forward Rules 6-6
6.1.5 Backward Rules 6-7
6.1.6 Pattern Matching 6-8
6.1.7 Guards 6-8
6.1.8 Theories 6-12
6.1.9 Tactics 6-12
6.1.10 Proof Construction 6-14
6.1.11 Guidelines for Proof 6-16
6.1.12 Ordering Rules in Theories 6-16
6.1.13 Using Guards in Rewrite Rules 6-17
6.1.14 Backwards versus Forwards Inference 6-17
6.1.15 Supplying Rules for Animation 6-17
6.1.16 Running Out of Memory 6-18

 ©1996 *B-Core (UK) Ltd.*

6.2 The POGenerator (pog) 6-19
6.3 The AutoProver (apr) 6-21
6.4 The InterProver (ipr) 6-23
6.4.1 The First Phase - Creation of Lemmas 6-24
6.4.2 The Second Phase - Proof of Lemmas 6-27
6.4.3 The PROOFMETHOD file 6-28
6.5 The Proof Printer (ppf) 6-31
6.6 ShowUnproved (sup) 6-32
6.7 ResetLevel (rsl) 6-33

7 Generators **7-3**
7.1 The Base Generators (gbo) (gbm) 7-3
7.1.1 GLOBAL declarations 7-6
7.1.2 BASE declarations 7-6
7.2 The Enumerator (enm) 7-9
7.3 The Interface Generator (itf) 7-11
7.3.1 The non-Motif Interface 7-12
7.3.2 The Motif Interface 7-13

8 Code **8-3**
8.1 The Translator (trl) 8-3
8.1.1 The ‘main’ Implementation: 8-3
8.1.2 The Translation Process 8-4
8.1.3 Note: 8-5
8.2 The Linker (lnk) 8-6
8.3 Run (run) 8-7

9 Documents **9-3**
9.1 The Document Mark-Up Tool (dmu) 9-3

 ©1996 *B-Core (UK) Ltd.*

9.1.1 Comments and Annotations 9-3
9.1.2 Positioning of Comments 9-4
9.1.3 Positioning of Newlines 9-4
9.1.4 Rules concerning Annotations 9-6
9.1.5 If LaTeX complains 9-6
9.1.6 The Document Options 9-7
9.2 Documents 9-8
9.3 Show (shw) 9-10
9.4 Print (prt) 9-11

10 Libraries **10-3**
10.1 The System Library (SLIB) 10-3
10.1.1 The Role of the System Library 10-3
10.1.2 Naming of Library Machines and their Operations 10-3
10.1.3 Library Machines 10-4
10.1.4 Multiple Objects Machines 10-4
10.1.5 Machine Rename_set_obj 10-5
10.1.6 Machine Rename_seq_obj 10-6
10.1.7 Machine Rename_str_obj 10-7
10.1.8 Machine Rename_fnc_obj 10-9
10.1.9 Machine Rename_ffnc_obj 10-9
10.1.10 Programming Concepts Machines 10-10
10.1.11 Machine Rename_Vvar 10-11
10.1.12 Machine Rename_Nvar 10-12
10.1.13 Machine Rename_Varr 10-12
10.1.14 Machine Rename_Narr 10-13
10.1.15 Mathematical Concepts Machines 10-14
10.1.16 Machine Rename_set 10-15

 ©1996 *B-Core (UK) Ltd.*

10.1.17 Machine Rename_Vseq 10-15
10.1.18 Machine Rename_Nseq 10-16
10.1.19 Machine Rename_Vfnc 10-17
10.1.20 Machine Rename_Nfnc 10-18
10.1.21 Machine Rename_Vffnc 10-19
10.1.22 Input Output Machines 10-19
10.1.23 Machine basic_io 10-20
10.1.24 Machine Rename_token_io 10-20
10.1.25 Machine Rename_file_io 10-20
10.1.26 Machine file_dump 10-20
10.1.27 Type Machines 10-21
10.1.28 Machine Bool_TYPE 10-21
10.1.29 Machine String_TYPE 10-21
10.1.30 Machine Scalar_TYPE 10-21
10.1.31 Machine Bit_TYPE 10-22
10.2 The Team Library (TLIB) 10-23
10.2.1 Making a New Project Team Library 10-23
10.2.2 Exporting Developments to the Team Library: 10-23
10.2.3 Committing Developments from the Team Library: 10-24
10.3 CreateSLIB 10-25
10.4 ExportTLIB 10-27

11 Utilities **11-3**

11.1 Reset 11-3
11.2 Find 11-4
11.3 Palettes 11-5
11.4 SaveDevelopment 11-6
11.5 BUnlock 11-7

 ©1996 *B-Core (UK) Ltd.*

11.6 BProcessInfo 11-8

12 References and Index **12-3**

 ©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

B-TOOLKIT USERS' MANUAL

SECTION 1

The B-Toolkit

©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

Figure 1: The B-Toolkit environment creation panel

1 The B-Toolkit

1.1 Getting Started

Before using the B-Toolkit, environment variables such as `BSITECODE` and `BKIT` must be correctly set. See Section 1.2.13 and Section 1.4.

The B-Toolkit is invoked by issuing the command `$BKIT/BToolkit` (or just `BToolkit` if the directory `$BKIT` is in the current path) from a suitable development directory — any directory in which you have read-write permission and which you intend to be the main directory for all new specification, design and implementation source files.

See figure 1.

If the toolkit is invoked for the first time in a directory (or where there is no existing development environment), a Motif panel appears to ask the user whether a new environment should be created. If the reply is Cancel then the toolkit is exited; if the reply is `OK` a new development environment is created, consisting of a set of development directories and configuration files. (See Section 1.2.1).

1.1.1 The Windows

The Motif interface to the B-Toolkit consists of two windows: the session window (the window from which the toolkit is invoked) and the B-Toolkit Motif panel (which is created). All interaction takes place in the main panel.

On start-up, the B-Toolkit Motif panel appears as portrayed in Figure 1.1.1. The panel is divided into a number of distinct areas as follows, starting from the top:

- The `Top Bar` which gives access to a number of general facilities.
- The `Information Panel` providing information on the development directory, and

 ©1996 *B-Core (UK) Ltd.*

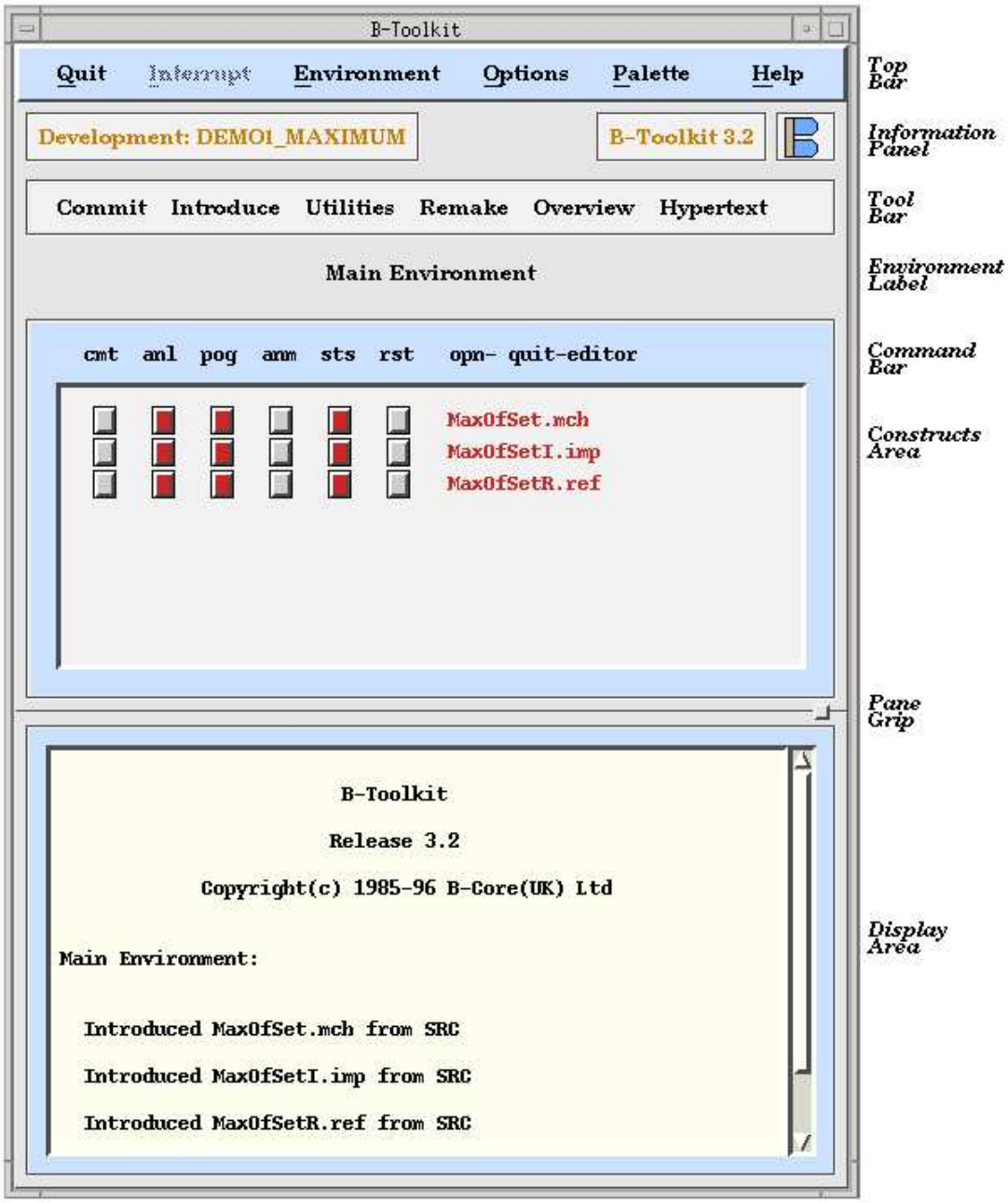


Figure 2: The B-Toolkit main panel

 ©1996 *B-Core (UK) Ltd.*

information on B-Core(UK) and B.

- The `Tool Bar` through which various tools can be invoked.
- The `Environment Name` which indicates the current environment (initially the Main Environment).
- The `Constructs Area` which lists, down the right-hand side, all of the constructs currently under configuration management. The term "construct" is used to refer to the different types of Abstract Machine (machine, refinement or implementation) and also to "system bases", "set enumerations", "interfaces", "documents", "proof files" and so on. A number of different files may be associated with each construct. Down the left-hand side are columns of buttons for the invocation of various commands on the corresponding construct. These commands are indicated in the `Command Bar` above the constructs area. Only those commands whose buttons are in full colour can be invoked; this serves to indicate, also, the status of each construct.
- The `Display Area` which serves to display messages from the toolkit relating to its current activity.

The `Constructs Area` and `Display Area` together form a paned window; their relative sizes can be altered by vertically sliding the grip that appears on the right between them.

1.1.2 The Mouse

Only the left-hand mouse button is used in the B-Toolkit interface. It is used to select menus, tools and commands by a single click. Clicking on many of the main panel entities will also bring up on-line help information. Double-clicking on items in selection boxes is a short-cut for a single click on that item, followed by clicking on the OK button to perform the default action.

1.1.3 Subsidiary Panels

Some of the commands activated from the Main panel popup a subsidiary panel associated with output from the B-Platform (see section 6). This panel displays information as the tool is executed, after which it is popped down leaving a copy of the display in the Display area in the Main panel.

Other panels are described under the appropriate tool descriptions.

1.1.4 The Top Bar

Except for Quit and Interrupt, each item on the Menu bar activates a pull-down menu. They are selected either by clicking with the mouse, or from the keyboard by using the

"command" key (marked with a diamond) simultaneously with the underlined letter in the menu name (its mnemonic).

Whilst a pull-down menu is activated, little other screen or keyboard action is permitted other than to select an item on that menu with the mouse. A menu can be deactivated by clicking elsewhere on the Menu bar. All pull-down menus may be "torn off" and displayed in a separate window (the resource setting `tearOffModel` should be set to `TEAR_OFF_ENABLED` in the XBMotif resource file).

- Quit
  Quits the B-Toolkit (via a confirmation dialoque panel).
- Interrupt
  Interrupts the current process. This Menu item is only active when other tools are running.
- Environment
  Selects B-Toolkit environments. See Section 1.2.8.
- Options
  Displays the values of various options that affect the operation of the B-Toolkit. These are described in detail in section Options.
- Palette
  Displays palettes of various language constructs to assist in writing AMN. See Section 11.2.
- Help
  Displays the Help menu. See Section 1.1.5.

1.1.5 The Tool Bar

The Tool bar allows access to a number of global tools accessible from all environments.

- `Commit`
  This pops up a selection box listing all constructs available for committing, allowing multiple commits to be performed. The commit command (`cmt`) in the Constructs Area only commits single constructs. A construct is available for committing when it is open for editing, and has been changed. You can exit from the Commit tool by clicking on Cancel.
- `Introduce`
  Serves to introduce constructs into the Constructs area (and thus under configuration management). New constructs can be introduced from a number of sources.

 ©1996 *B-Core (UK) Ltd.*

1.1.6 On-line Help

Extensive on-line help information is available through the `Help` menu, which displays a list of subjects. Selecting a subject calls up an editor window to display the corresponding text.

Other on-line help can be obtained by clicking with the mouse on many of the entities on the main panel. (Try clicking, for instance, on the environment name.)

©1996 *B-Core (UK) Ltd.*

1.2 The B-Toolkit

The B-Toolkit comprises a suite of fully integrated software tools designed to support a rigorous or formal development of software systems using the B-Method. The tools are accessed through a custom build X Windows Motif Interface, providing full on-line help facilities.

The B-Method uses the notion of Abstract Machines to specify and design software systems. Abstract Machines are specified using the Abstract Machine Notation (AMN) which is in turn based on the mathematical theory of Generalised Substitutions.

The Toolkit also provides a development environment automating the management of all associated files, ensuring that the entire development, including code and documentation, is always in a consistent state. One aspect of this environment is that the effect of change in a single file on the entire system is communicated to the user before the decision is made to commit the change to the system (this information may also be ascertained through the `Status` tool for machines, refinements and implementations).

Currently the Toolkit provides the following tools for supporting software development:

1. A specification, design and code configuration management system, including integrity and dependency management and source file editing facilities.
2. A set of software specification and design analysis tools, which includes syntax checkers, type checkers and a specification animator.
3. A set of verification tools, which includes a proof-obligation generator and automatic and interactive provers, together with a proof printer for displaying proof obligations/proofs.
4. A set of coding tools, which includes a translator, linker, rapid prototyping facilities and a reusable specification/code module library.
5. Facilities to supplement the reusable library.
6. A set of generators providing facilities to generate: base object AMN specification/implementations (the latter implemented on the reusable library) from simple declarative descriptions of structured data, enumerators providing i/o facilities for enumerated sets, and interfaces (either specification/implementations - again built on the reusable library, or alternatively on the X/Motif library) to provide a run time environment.
7. A documentation tool for automatically producing fully cross-referenced and indexed type-set documents from source files, together with facilities to display on screen or send to a printer.

 ©1996 *B-Core (UK) Ltd.*

8. A re-making tool for automatically re-checking and re-generating specifications, designs, code and documentation after modifications to source files.

9. A hypertext tool providing, in hypertext form, the facility to navigate through an entire development; each hypertext construct is automatically built from its Abstract Machine counterpart, allowing constructs external to the development (for example, informal requirements) to be hypertext-linked to the formal development.

10. An overview tool providing, in picture form, the facility to navigate through an entire development via a Motif push- button interface; it provides views of both specification construction and design, enabling the user to see a complete picture of the development, or to zoom in on a particular aspect of it.

Before using the toolkit, the mandatory environment variables BSITECODE, BLIC, and BKIT must be correctly set (see the 'B-Toolkit Installation Guide'); further environment variables aid the customisation of the toolkit (BEDIT - see Help for Editor, BCC - see Help for Translator, and BSDVI and BPDVI - see Help for Document).

The B-Toolkit is invoked by issuing the command BToolkit (this script residing in the $BKIT directory) from a suitable development directory, any directory in which you have read-write permission.

1.2.1 Development Environment

If the toolkit is invoked where there is no existing development environment (which will be the case when it is invoked for the first time in a directory), the user is asked whether a new environment should be created. If the reply is 'no' then the Toolkit is exited; if the reply is 'yes' a new development environment, consisting of a set of development directories and configuration files, is created. The following directories are created if they do not already exist:

- `SRC`
  for source files
- `CFG`
  for files committed to configuration
- `ANL`
  for the intermediate analysed form
- `TYP`
  for TypeChecker information

 ©1996 *B-Core (UK) Ltd.*

- `POB`
  for the proof obligation files
- `HTX`
  for Hypertext files
- `TEX`
  for typeset document files
- `PMD`
  for the proof method files
- `CDE/C`
  for C-program files

1.2.2 The Motif Interface

See figure
1.1.1.

The Motif Interface to the B-Toolkit comprises a Top Bar, an Information Panel, a Tool Bar, a Constructs Area displaying the constructs and the currently available options, and a Display Area displaying messages from the toolkit.

The Current Directory, Release Information and information on B are displayed as push-buttons in the Information Panel (invoking them will popup help windows displaying appropriate information). The Environment Label is located immediately below the Tool Bar (clicking on it will popup a help window describing the current environment).

The Command Bar is incorporated in the top frame of the Constructs Area; clicking on any of the labels will pop up a help window providing a brief description of each option (including its applicability).

The relative heights of the Constructs Area and Display Area may be varied by dragging the pane grip (situated on the right between the two windows) up or down.

The interface incorporates many popup windows (for sub-menus, option selection, confirmation and so on) each incorporating a 'Help' button providing a further popup containing help appropriate to that action), in addition to the main Help facility on the Top Bar. Selection(s) may be made by clicking on the appropriate item(s) and selecting 'OK'; alternatively it is possible to 'double-click' on an item (or the last item in the case where more than one is being selected). 'Cancel' dismisses the popup window.

The appearance of the toolkit may be customised by editing the XBMotif resources file (you should seek the guidance of your system administrator if you wish to do this).

Each of the tools mentioned below in outline is described in more detail elsewhere.

 ©1996 *B-Core (UK) Ltd.*

1.2.3 Constructs Area

The term 'construct' is used to refer to the different types of Abstract Machine (machine, refinement or implementation) and also to 'system bases', 'set enumerations', 'interfaces' or 'documents'. A number of different files may be associated with each construct. Constructs are listed down the right hand-side of the Constructs Area. Different types of construct are displayed in different B-Toolkit environments. Colour-coding is used in the Constructs Area and Overview Browser to indicate the origin of each construct :-

- **Red** is used for those constructs introduced by the user.
- **Blue** is used for library constructs.
- **Green** is used for those constructs generated by tools such as the Base Generators and Interface Generator.

On the left-hand side of the Constructs Area are rows of command buttons which are used to invoke the various tools on each construct. Active buttons are coloured, inactive are grey. The status of each construct is manifest by the combination of its command buttons that are active.

1.2.4 The Top Bar

The Top Bar provides the following facilities:

- `Quit` (quits the Toolkit)
- `Environment` (selects another Environment)
  - Main
  - Provers
  - Generators
  - Translators
  - Documents
- `Options` (provides a menu of flags)
  - Remake
    - POGenerator (included/excluded)
    - AutoProver (included/excluded)
    - ProofPrinter (included/excluded)

 ©1996 *B-Core (UK) Ltd.*

        * DocumentMarkUp (included/excluded)
        * Translator (included/excluded)
        * Linker (included/excluded)
        * ITFGenerator (included/excluded)
    - Construct Display
        * Library constructs (show/hide)
        * Generated constructs (show/hide)
    - Editor
        * Commit edits (preserve editor/kill editor)
    - Documents
        * Labels (on/off)
        * Clause cross-references (on/off)
        * Construct cross-references (on/off)
        * Index (on/off)
        * Point size (10/11/12)
    - Programming Language
        * Programming language (C/ANSI C)
    - DMU Language
        * Document mark-up language (LaTeX)
    - Interface
        * Interface type (Motif/Non-Motif)

- `Palette` (provides a menu of palettes)
    - Bases
    - Documents
    - Enumerations
    - Implementations
    - Interfaces
    - Machines
    - Ops_Imp
    - Ops_Mch
    - Ops_Ref
    - Refinements

- `Help` (provides a menu for help)

1.2.5 The Information Panel

The Information Panel provides popups which give:

- Information on the development directory
- Information on B-Core(UK)
- Information on B

1.2.6 The Tool Bar

The Tool Bar provides the following facilities:

- Commit: (providing a menu of all currently open edited constructs for committing the edits)
- Introduce: (enabling new construct templates to be introduced)
  - Introduce new construct: (provide a new construct template)
    - Machine
    - Refinement
    - Refinement of analysed construct
    - Implementation
    - Implementation of analysed construct
    - Base
    - Enumeration
    - Enumeration of analysed machine(s)
    - Interface
    - Interface of implemented machine
    - Document
    - Document of configured constructs
  - Introduce construct(s) from SRC (provides a menu of SRC constructs)
  - Introduce construct from SLIB (provides a menu of SLIB constructs)
  - Introduce construct from TLIB (provides a menu of TLIB constructs)
- Utilities:
  - Shell (opens a shell tool in the current directory)
  - Find: (searches for the occurrence/usage of an identifier)

 ©1996 *B-Core (UK) Ltd.*

  - Rename construct (changes a construct's name/filename)
  - Remove construct(s) (removes construct(s) from configuration)
  - Reset all constructs (enables the entire development to be reset)
  - Reset all documents (enables all marked-up constructs to be reset)
  - Reset all code (enables all code to be reset)
  - Save Development (saves the current state of the development)
  - Remove unused LIB constructs (removes all unused library constructs)
  - Create SLIB construct (create a new reusable SLIB construct)
  - Export construct to TLIB (export a sub-development to TLIB)
- Remake: (development rebuilt depending on 'Remake Options' settings)
- Overview (enables a design/specification overview, or that of a specific construct)
  - Design Overview
  - Specification Overview
  - Construct
- Hypertext (enables navigation through a development via automatically-generated hypertext equivalents of analysed constructs)
- Close: (providing a menu of all currently open constructs for closing)

1.2.7 The Display Area

The Display Area is used to display messages and general output from the various tools.

Some of the main tools, such as the Analyser and Proof Obligation Generator, popup a subsidiary window to receive their output, and this output is copied into the Display Area when the job is complete and the window is popped down.

1.2.8 The B-Toolkit Environments

The environments are accessed through the Environment Menu on the Top Bar. A command for a construct is applicable when the relevant command button is coloured. Each environment contains the commands:

- `cmt` (commits the edits of a currently-open construct)
- `opn -quit-editor` (a toggle for opening an edit/quitting an edit)

 ©1996 *B-Core (UK) Ltd.*

1.2.9 The Main Environment

All constructs (except proofs, documents and ops) are displayed in this environment. Display may be filtered by changing the 'Construct Display' settings through the Top Bar Options button.

- `anl` (invoke the Analyser)
- `pog` (invoke the Proof Obligation Generator)
- `anm` (invoke the Animator)
- `sts` (display status)
- `rst` (reset construct)

1.2.10 The Provers Environment

Constructs which have been POGenerated, together with proof files, are displayed in this environment.

- apr (invoke the AutoProver)
- ipr (invoke the InterProver)
- pmd (edit PROOFMETHOD file)
- rsl (reset to a previous a proof level)
- ppf (print proof)
- sup (show undischarged proof obligations)
- `pob` (number of currently undischarged proof obligations)
- `tot` (total number of proof obligations)

1.2.11 The Generators Environment

Base Object Machine, Enumeration, Base Object Operation and Interface constructs are displayed in this environment.

For base/ops objects:

- gbo (generate base object operations)

 ©1996 *B-Core (UK) Ltd.*

- gbm (generate base object machines/implementations)

For enumerations:

- enm (invoke the enumeration specification/code generator tool)

For interfaces:

- itf (generate specification/implementation or Motif interface)

1.2.12 The Translators Environment

Implementations that have been analysed, or Motif interfaces that have been generated are displayed in this environment.

- trl (invoke the Translator tool)
- lnk (invoke the Linker)
- run (run code)

1.2.13 The Documents Environment

All constructs that are already marked up, or are ready for marking up (machines, refinements or implementations currently analysed; bases, enumerations or interfaces currently generated; proofs that have been constructed) are displayed in this environment.

Display may be filtered by changing the 'Construct Display' settings through the Top Bar Options button.

- dmu (mark-up document)
- shw (preview document)
- prt (print document)

 ©1996 *B-Core (UK) Ltd.*

1.3 Installation

This document describes

- The contents of the installation disks.
- How to install the B-Toolkit Release 3.1.
- The run-time library requirements of the B-Toolkit.
- How to install the licence file and set the `BSITECODE`.
- How to run the B-Toolkit.
- How to increase the memory capacity of the B-Toolkit by recompiling the B-Platform.

1.3.1 Contents of the 5 Disks

The content of the 5 disks comprises the B-Toolkit in compressed form in the following structure:

- `install.BToolkit` is used to install the toolkit under `OpenWin` or the `X Window System`.
- `BKIT` is the B-Toolkit directory, containing:
  - `BHELP`
    a directory containing on-line help files.
  - `BLIB`
    the B-Toolkit library directory.
  - `BPALETTE`
    a directory containing on-line palette files.
  - `BPLAT`
    the B-Platform directory.
  - `BToolkit`
    the script that the user invokes to run the B-Toolkit.
  - `BUnlock`
    the Unlock facility. See section 11.4.
  - `BProcessInfo`
    the display B processes information facility. See section 11.5.
- `BDEMO` is a directory containing cases studies of Abstract Machine developments. Each demonstration has a `READ_ME` .

The disks do NOT contain the "B licence file", `B.lic`, which is necessary to run B-Toolkit, and which is separately supplied to every customer.

 ©1996 *B-Core (UK) Ltd.*

1.3.2 Installation

1.3.3 Reading the Disks

The first thing to do is to select an installation directory in which you have write permission. Make this directory the current directory, and extract into it the files from each disk in turn by repeatedly using the command

```
tar  xvf  /dev/fd0
```

(or the command that is appropriate to your system).

You should then verify that the `BT_DISK_CONTENTS` directory is in your current directory.

1.3.4 Running the Install Script

The purpose of the installation is to load the directories `BKIT` and `BDEMO` in the current directory.

After successful installation of the toolkit, it is necessary to set various environment variables, as detailed below.

First you copy the files

```
btar0.gz  btar1.gz  btar2.gz  btar3.gz  btar4.gz install.BToolkit
```

from `BT_DISK_CONTENTS` into your current installation directory.

Then you run the installation script, by issuing the command:

```
./install.BToolkit
```

The installation script will ask you to verify that `cc` can be used for compilation and linking. You have the option to change to another C-compiler.

Suppose the selected installation directory is `/usr/local/tools/BT_3.1`. The installation proceeds as follows:

```
-----------------------------------------------

Script for installation of B-Toolkit Release 3.1
```

 ©1996 *B-Core (UK) Ltd.*

```
        ------------------------------------------------

--  Compile and link using:

--  cc

*****  OK (y/n)? y

--  Processing btar0 -- uncompressing ... extracting ... done.
--  Processing btar1 -- uncompressing ... extracting ... done.
--  Processing btar2 -- uncompressing ... extracting ... done.
--  Processing btar3 -- uncompressing ... extracting ... done.
--  Processing btar4 -- uncompressing ... extracting ... done.

--  Starting installation

        ....

--  Installation complete.



                     ---------------

                       B-Toolkit
                      Release 3.1
                      Information

                     ---------------




--   B-Toolkit Release 3.1 created:

        Mon Jan  8 12:10:52 GMT 1996



--  All B-Toolkit users should set
    the following environment variables:

--     BKIT should be set to /usr/local/tools/BT_3.1/BKIT:
```

 ©1996 *B-Core (UK) Ltd.*

```
          setenv BKIT /usr/local/tools/BT_3.1/BKIT

--     LD_LIBRARY_PATH should include Motif and X libraries

          setenv LD_LIBRARY_PATH /usr/lib:/usr/openwin/lib:/usr/dt/lib

--     BEDIT should be set to an appropriate editor, e.g.:

          setenv BEDIT "textedit -Wp  501 401 -Ws 632 500"

--     BSITECODE should be set:

          see your Installation Documentation

--     BLIC must be set to
          the directory containing the B licence file,
          B.lic, which is normally in $BKIT, eg:

          setenv BLIC /usr/local/tools/BT_3.1/BKIT


--  Additionally, LaTeX users should ensure the following:

--     BSDVI should be set to
       an appropriate LaTeX screen viewer, e.g.:

          setenv BSDVI xdvi

--     BPDVI should be set to
       an appropriate LaTeX print script, e.g.:

          setenv BPDVI /usr/local/bin/printdvi


--  Optionally, BCC may be set to override
    the default C-Compiler 'cc', eg:

          setenv BCC "gcc -g"
```

 ©1996 *B-Core (UK) Ltd.*

```
------------------

--  The above information is contained in the file:
    /usr/local/tools/BT_3.1/READ_ME

------------------
```

The above information from the installation script is saved in a `READ_ME` in the installation directory. It gives information about setting environment variables. See section 1.3.4 for more details.

1.3.5 Run-time Library Requirements

The B-Toolkit requires dynamic linking to various `X` and `Motif` run-time libraries.

You can find out about the B-Toolkit's requirements for run-time libraries by setting the `BKIT` variable as described above, and issuing the command:

```
ldd  $BKIT/BLIB/BMotif
```

If there are required libraries that are not found, then ask your system administrator to help you locate them, and set the environment variable `LD_LIBRARY_PATH` accordingly.

1.3.6 The B Licence File

The B Licence file, `B.lic`, and the Site Code are supplied separately. The licence file should be placed in an appropriate directory, normally, but not necessarily, in `$BKIT`.

The environment variable `BLIC` should be set to point to the directory in which BLIC is placed, e.g.

```
setenv BLIC /usr/local/tools/BT_3.1/BKIT
```

The environment variable `BSITECODE` should be set to the value of the Site Code supplied with the licence. This code looks like a credit-card number, with four groups of four digits separated by hyphens. For example:

```
setenv BSITECODE 1234-2345-3456-4567
```

 ©1996 *B-Core (UK) Ltd.*

1.3.7 Running the B-Toolkit

After setting the environment variables and positioning the B Licence file as described above, change to a convenient working directory (for example "/fs/mydir/tmp") and issue the command:

```
$BKIT/BToolkit
```

The following should appear in the shell window:-

```
                    B-Toolkit

                   Release 3.1

          (c) Copyright B-Core(UK) 1995
```

A panel should then appear containing the text:

```
Create new B-Toolkit directory:

  /fs/mydir/tmp
```

If you respond by clicking the mouse on the OK button, the shell window will display

```
Creating new B-Toolkit environment ... done
```

and, after a few seconds, the main B-Toolkit Motif panel will appear on the screen. This indicates correct functioning.

More information about running the B-Toolkit can be found in Getting Started , section 1.

1.3.8 Errors

If the installation fails, a message similar to the following will appear.

```
--  Installation aborted
< reason >
```

 ©1996 *B-Core (UK) Ltd.*

After correction of the problem, the installation should be restarted, by invoking the `install.BToolkit` again.

If the following error occurs:

```
--  Couldn't copy XBMotif resource file into /usr/lib/X11/app-defaults
```

you can proceed without re-installing by copying the file `BKIT/BLIB/XBMotif` into the home directory of B-Toolkit users.

1.3.9 Increasing the Capacity of the B-Toolkit

The capacity of the toolkit is dictated by that of the compiled bplatform. This capacity may be increased (but not decreased) by editing the file:

```
$BKIT/BPLAT/bplatsize.c
```

and increasing the value of `bplatform_size`; the bplatform may then be recompiled by issuing the command:

```
cc -O -o bplatform bplatsize.c bplatform.o
```

The maximum number of automatically-generated hypotheses during a single auto-proof step may also be changed by editing the same file and changing the value of `max_fwd_gen_hyp` and recompiling the bplatform as described above. It is not recommended that this figure be increased unless it is essential, since the speed of auto-proving will be adversely affected by large generated hypotheses.

 ©1996 *B-Core (UK) Ltd.*

1.4 The B-Toolkit Environment

The B-Toolkit requires certain Unix environment variables to be set for correct function, and can be configured through the use of certain others. These are detailed below. Examples are shown as if operating within the Unix C-Shell, `csh`.

1.4.1 Obligatory Environment Variables

- `BKIT`
  This must be set to the directory in which the B-Toolkit has been installed. For example:-

  ```
  setenv BKIT /usr/local/tools/BKIT
  ```

- `BSITECODE`
  This variable should be set to the 16-digit licence code supplied to you be B-Core (UK) Ltd.. For example:-

  ```
  setenv BSITECODE 1234-1234-1234-1234
  ```

- `BLIC`
  This specifies the directory in which the B-Toolkit licence file resides. For example:-

  ```
  setenv BLIC /usr/fred
  ```

1.4.2 Optional Environment Variables

- `BNOBELL`
  This allows you to turn off the bell sound the the B-Toolkit makes when it starts up and completes jobs. The default is on. To turn it off, use:

  ```
  setenv BNOBELL 1
  ```

- `BEDIT`
  This allows you to specify your own editor for use with the B-Toolkit. The default for SunOs is `xedit`, and for Solaris it is `"textedit -Wp 501 401 -Ws 632 500"`.

- `BHYPEDIT`
  This allows you to specify your own editor for viewing hypotheses during proof.

- `BHTML`
  This allows you to specify an HTML Browser for use with the B-Toolkit. The default is in a plain ASCII format. Other options might be, for instance:-

 ©1996 *B-Core (UK) Ltd.*

```
setenv BHTML Mosaic

setenv BHTML "xterm -geometry 80x75 -e lynx"
```

Note that the help makes use of GIF files for display images. By default Mosaic and Netscape use 'xv' for viewing these images, but some other GIF viewer may be available at your site.

- BCC
  This variable allows you to specify the C compiler to use with the translators of the B-Toolkit, and to add additional options, such as for debugging (although, of course, you will never need that!). For example:-

  ```
  setenv BCC "gcc -g"
  ```

  The default is `cc`.

- BSDVI
  This allows you to specify the command to be used for displaying marked-up documents on the screen, for example:-

  ```
  setenv BSDVI "xdvi -s 4"
  ```

- BPDVI
  This allows you to specify the command to be used for printing marked-up documents on the screen, for example:=

  ```
  setenv BPDVI "printdvi"
  ```

  where `printdvi` is the script:-

  ```
  #! /bin/csh -f
  dvi2ps $1 > $1.ps
  lpr $1.ps
  lpq
  ```

- BTEAMLIB
  This specifies the path location of the Team Library. e.g.

  ```
  setenv TEAMLIB /usr/fred/BTeamLib
  ```

 ©1996 *B-Core (UK) Ltd.*

1.4.3 Other Environmental Considerations

One other environment variable affects the operation of the B-Toolkit:-

- `PATH`
  If operating under Solaris, the `PATH` variable may have to be modified to allow the B-Toolkit to access compatible versions of various Unix commends, such as `ps`. The directory `/usr/ucb` should be placed *before* `/usr/bin` in the `PATH` variable.

  It may also be helpful to place `\$BKIT` in `PATH`.

  For example:-

  ```
  setenv PATH $BKIT:/usr/ucb:${PATH}
  ```

Other variables local to LaTeX or DVI, such as XDVIFONTS, may also have to be set correctly, and for this you such consult your local support team.

Another consideration are the Motif defaults for the B-Toolkit, which the user can configure by editing the XBMotif file, which normally resides in the user's home directory.

 ©1996 *B-Core (UK) Ltd.*

1.5 Options

The Options menu is available on the Top Bar, and allows the user to customise various aspects of the B-Toolkit. Each option may be 'toggled' by clicking on the current setting.

1.5.1 Remake

When a system Remake has been activated (through the Tool Bar) the extent of the Remake is determined by the current settings of the Remake Options. Each of the following tools may be 'included' in or 'excluded' from a Remake through the option setting:

```
POGenerator
AutoProver
ProofPrinter
DocumentMarkUp
Translator
Linker
ITFGenerator
```

The initial setting is 'included' for each tool.

1.5.2 Construct Display

In large developments, it may be desirable to 'hide' library and/or generated constructs from being displayed in the Constructs Area. (Even though a construct is not displayed, it is still, of course, present in the development.) This may be achieved through toggling the 'show'/'hide' option setting:

```
Library constructs
Generated constructs
```

The initial setting is 'show' for both.

This option is applicable only to the Main and Documents Environments.

1.5.3 Editor

When an edited construct has been committed, the editor is either dismissed or retained according to the 'kill editor'/'preserve editor' setting:

```
Commit edits
```

 ©1996 *B-Core (UK) Ltd.*

The initial setting is ‘kill editor’.

1.5.4 Documents

These options determine the output from the DocumentMarkUp tool:

- Labels:
  When this option is set to ‘on’, a ‘\label’ is created in the tex file for each newly-introduced variable, set, constant and operation name as it is being marked up, allowing the user to use ‘\ref’ and ‘\pageref’ inside annotations for MACHINES, REFINEMENTS and IMPLEMENTATIONS. The initial setting is ‘on’.

- Clause cross-references:
  When this option is set to ‘on’ cross references to set, constant and operation names (if any) will automatically be produced at the end of the the PROPERTIES, INVARIANT, ASSERTIONS and INITIALISATION clauses, and following each operation of a MACHINE, REFINEMENT and IMPLEMENTATION. The initial setting is ‘on’.

- Construct cross-references:
  When set to set to ‘on’ a construct cross reference to all variable, set, constant and operation names (if any) is produced at the end of each MACHINE, REFINEMENT and IMPLEMENTATION. The initial setting is ‘on’.

- Index:
  When this option is set to ‘on’, an index is automatically produced at the end of each DOCUMENT, giving page references to all occurrences of variable, set, constant and operation names (if any). The initial setting is ‘on’.

- Point size:
  May be set to ‘10’, ‘11’ or ‘12’. The initial setting is ‘10’.

For more information, see DocumentMarkUp (page 3).

1.5.5 Programming Language

The language may currently be set to ‘C’ or ‘ANSI C’. The initial setting is ‘C’.

1.5.6 DMU language

Currently the only document mark up language is ‘LaTeX’.

 ©1996 *B-Core (UK) Ltd.*

1.5.7 Interface

Motif and non-Motif interfaces may be generated (see Interface , page 11, for an explanation). The initial setting is 'Motif'.

 ©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

B-TOOLKIT USERS' MANUAL

SECTION 2

The B-Method

©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

2 The B-Method

2.1 Introduction

Formal methods of software engineering offer a number of benefits. Firstly, they enable the production of reliable software systems for which all possible behaviour can be accurately predicted and verified to meet their functional specifications. Secondly, they have the excellent potential for automation, which will reduce software production costs. Thirdly, they encourage accountability, uniformity of approach and reusability from specification through to coding, all easing the task of software maintenance.

Existing formal methods have failed to make provision for all of these benefits, and existing applications of formal methods have thus failed to demonstrate the advantage of the approach convincingly. So far, the emphasis has been on formal specification, rather than on the entire software cycle of specification, coding and maintenance.

The B-Method [5],[23],[4],[6],[8] is a formal software development process which has been designed as a complete methodology for production of highly reliable, portable and maintainable software on an industrial scale; extremely powerful integrated tool support is provided at all stages of the development process, the B-Toolkit.

This paper is based on an earlier paper written in collaboration with J-R Abrial ([3]).

2.2 The B-Method and the B-Toolkit

Several important principles have guided the design of the B-Method:

1. The B-Method is a mathematical method that belongs to the "model oriented" approach to software construction (VDM [19] [20] and Z [25] employ a similar approach).

2. Great attention has been paid in ensuring that the notational aspect of the method is as simple as possible. In particular, there is no real distinction between the specification notation and the programming notation; the latter appears as a restricted subset of the former.

3. The method is founded on set theory in a way which is made as solid as possible by reconstructing the original Zermelo set theory [27],[28] within the method itself. In particular, each construct of the specification (and programming) notation has a set-theoretic counterpart.

4. The method has been designed in parallel with the B-Toolkit that supports it, so that every aspect of the method is supported by providing significant help to the user.

 ©1996 *B-Core (UK) Ltd.*

5. Separate techniques are proposed within the method to support the design of (large) specification construction and that of (large) program construction.

6. The method is based on a series of embedded notations: the logical notation, the basic set notation, the relational notation, the mathematical object notation, the generalised substitution notation, and, at the highest level, that of the Abstract Machine.

The notation used by the B-Method relies on an extension to Dijkstra's calculus [16]. The extension allow specification of operations in terms of preconditions and postconditions, and permits object-oriented designs. Similar extensions are common within university research, eg. Morgan's Specification Statement [24].

This paper assumes a knowledge of the underlying logical and set notations, and describes some aspects of the Abstract Machine Notation (AMN) and the Generalised Substitution Language (GSL); a full description of AMN and GSL may be found in Section 3.

J-R. Abrial has given a set theoretic model for the notation to ensure its soundness, and his book is due to be published later this year [2].

The B-Toolkit which supports the B-Method is an integrated suite of computer programs which covers many aspects of software development, including:

1. Full configuration and dependency management of all source files, including abstract specification documents as well as binary code files.

2. Syntax and type-checking of specification documents as well as low level design documents, with comprehensive error reporting .

3. Verification condition generation (which generates the proof-obligations necessary in order to guarantee specification consistency and correctness of refinement).

4. Automatic & interactive provers for discharging the verification conditions.

5. Specification animation, enabling the specification to be 'run' and tested.

6. Code generation from declarative descriptions, facilitated by a re-usable library of code modules.

7. A Library of reusable code modules which integrate automatically with models from the reusable specification library.

8. A translator for translating low level design documents into C.

9. Rapid prototyping, facilitated by an interface generator, built on the re-usable library.

10. Automatic Markup and Indexing of documentation.

11. Remaking facilities when a source file is altered; the scope of the remake is determined by the toolkit, and the system is rebuilt (if possible automatically) to its former state, including code and documentation.

12. A multi-user facility for team software development.

13. All the individual tools are integrated and designed to run automatically.

14. All tools are activated from a simple to use window-based control panel.

A more detailed description of the B-Tookit components is given in section 2.7.

2.3 Abstract Machine Notation

In the B-Method, specifications, refinements and implementations are presented as Abstract Machines. Clauses within the Abstract Machine Notation declare a context of global constraints, given constants, abstract sets and their properties; further AMN clauses provide a list of abstract state variables and operations on those variables, the latter having optional preconditions, parameters and results. The operations provide the interface for the outside world to the state variables encapsulated within the Abstract Machine.

Abstract Machines may be parameterised (with possible parameter constraints) so that instances of machines can be reused in the incremental construction of more complex machines.

2.3.1 The Generalised Substitution Language

In order to describe operations, a "before-after" predicate for each operation could be used, as done in VDM [20] and in Z [25]. However, the B-Method uses instead an extension of the notion of substitution (as used in mathematics) called the Generalised Substitution Language (GSL) for specifying an operation in a dynamic systems. AMN is an extended form of GSL, itself a generalisation of Dijkstra's guarded command notation [16], enhanced with preconditioning and unbounded choice, similar to the generalisations proposed in [24].

The statements in AMN (and GSL) are very concise and easy to read; there is no difference between the specification and programming notation; the generalised substitution notation, in common with all programming notations, does not refer to variables which do not change.

Each generalised substitution S is defined as a predicate transformer which transforms a postcondition R into the weakest precondition for S to establish R. We write this condition as $[S]R$ and it is read : S establishes R. S may take one of the following forms: the simple substitution ($v := e$); the `skip` substitution; the *pre-conditioned* substitution (|);

 ©1996 *B-Core (UK) Ltd.*

the *bounded choice* substitution ([]); the *guarded* substitution ($\Longrightarrow$); the *unbounded choice* substitution (@).

These generalised substitutions are axiomatised:

GSL Syntax	Definition
$[v := e]\, R$	R *with* v *replaced by* e ($free\ occurrences\ of$ v *are replaced*)
$[\texttt{skip}]\, R$	R
$[P \mid S]\, R$	$P \ \wedge\ [S]\, R$
$[P \Longrightarrow S]\, R$	$P \Rightarrow [S]\, R$
$[S[]T]\, R$	$[S]\, R \ \wedge\ [T]\, R$
$[@l.S]\, R$	$\forall\, l\ .\ [S]\, R$, if l is not free in R

Note that, thanks to the two choice substitutions, it is possibe to specify the behaviour of a dynamic systems in a non-deterministic way: this is important, since a specification not always should give the precise and determinate details of a future systems. Such details will be added later in the refinement/implementation phases leading eventually to code.

Below, are examples of syntactic rewriting rules (sugars) allowing us a very rich "pseudo-programming" specification notation (AMN).

 ©1996 *B-Core (UK) Ltd.*

AMN Substitution	GSL Substitution
BEGIN S **END**	S
PRE P **THEN** S **END**	$P \mid S$
CHOICE S **OR** $\cdots$ **OR** T **END**	$S \; [] \; \cdots \; [] \; T$
IF P **THEN** S **ELSE** T **END**	$P \Longrightarrow S \; [] \neg P \Longrightarrow T$
IF P **THEN** S **END**	$P \Longrightarrow S \; [] \neg P \Longrightarrow \texttt{skip}$
ANY z **WHERE** P **THEN** S **END**	$@z \; . \; (P \Longrightarrow S)$
VAR z **IN** S **END**	$@z \; . \; S$
$x \; := \texttt{bool}(P)$	$P \Longrightarrow x := \texttt{TRUE} \; [] \neg P \Longrightarrow x := \texttt{FALSE}$

2.4 Abstract Machine Specification

Proof of internal consistency of an Abstract Machine specification requires demonstration that its context may exist - the formal parameters, constants and variables - and that within this context of the machine, the initialisation establishes the invariant, and each machine operation maintains that invariant. For example, the following Abstract Machine:

©1996 *B-Core (UK) Ltd.*

MACHINE

Machine_name(x)

CONSTRAINTS

P

CONSTANTS

y

PROPERTIES

Q

VARIABLES

z

INVARIANT

R

INITIALISATION

T

OPERATIONS

Operation_name $=$ **PRE** L **THEN** S **END ;**

$\ldots$

END

gives rise to the following proof obligations:

$$\exists x \,.\, P$$

$$P \;\Rightarrow\; \exists y \,.\, Q$$

$$(P \;\wedge\; Q) \;\Rightarrow\; \exists z \,.\, R$$

$$(P \;\wedge\; Q) \;\Rightarrow\; [T]R$$

$$(P \;\wedge\; Q \;\wedge\; R \;\wedge\; L) \;\Rightarrow\; [S]R$$

©1996 *B-Core (UK) Ltd.*

In the above, $[S]R$ is read as "S establishes R"; the first three are concerned with consistency of contextual information (i.e if these conditions are not met then we have given a contradictory specification, which we should not attempt to implement); the fourth concerns the initialisation, and checks whether the invariant condition is satisfied initially; the last proof obligation concerns the operations (there will be one such proof obligation for each operation of the machine), and this checks whether an operation maintains the invariant condition.

2.4.1 Structuring and Constructing Specifications

In order to manage the complexity of the specification process it is essential that specifications can be constructed and verified in an incremental way, and that existing specifications can be used to build new specifications. Whilst in the construction of programs we are concerned with separate compilation, in the construction of large specifications from component parts we are concerned with separate verification and proof.

The key to the problem is a new generalised substitution construct, the multiple substitution ($\|$), allowing us to put together two substitutions working on distinct variables. Thus we extend the syntax of GSL to permit multiple substitution, and we may now derive such laws as:

GSL Syntax	Equivalent Syntax
$S \parallel \texttt{skip}$	S
$S \parallel (P \mid T)$	$P \mid (S \parallel T)$
$S \parallel (T \,[]\, U)$	$(S \parallel T) \,[]\, (S \parallel U)$
$S \parallel (P \implies T)$	$P \implies (S \parallel T), \quad if\ S\ \text{"}terminates\text{"}$
$S \parallel @z \cdot T$	$@z \cdot (S \parallel T), \quad if\ z\ is\ not\ free\ in\ S$

From these properties we can show that multiple substitution ($\|$) always be eliminated in any formal reasoning about AMN operations.

 ©1996 *B-Core (UK) Ltd.*

We correspondingly extend AMN such that it is now extended by the five Abstract Machine Clauses described in the two sections that follow.

2.4.2 The INCLUDES, PROMOTES and EXTENDS Clauses

These concerns are provided for by employing what is known as the *semi-hiding principle*: a variable from one Abstract Machine can be included within another Abstract Machine but not modified from within that Abstract Machine (and so the invariant of an included machine does not add to the proof obligations of the machine in which it is included). The included variables are thus "read-only", but may appear in the invariant of the including machine. The only way in which the included variables may be changed (or read) is by "calling" one of the included operations from within the body of an operation of the including machine.

AMN allows for the inclusion of existing Abstract Machine specifications with distinct variables to be included to build new specifications, following the semi-hiding principle by the use of the INCLUDES clause. Machine parameterisation and renaming permit particular instances of different machines, as well as distinct copies of the same machine, to be included within a new machine. The variables of the new machine include the variables from each of the included machines; the invariant of the new machine includes the conjunction of the invariants of all included machines; the initialisations from the included machines are also inherited. New variables can be introduced and new invariant conditions (a "gluing invariant") can be imposed; each operation of an included machine is accessible to each operation of the new machine. The INCLUDES clause is transitive.

The PROMOTES clause contains the list of operation names of those operations of the included machines that become, without any alteration, operations of the new machine.

The EXTENDS clause has exactly the same form as the INCLUDES clause: all extended machines are included and, moreover, all operations of extended machines are automatically promoted to the new machine.

2.4.3 The USES and SEES Clauses

While the INCLUDES mechanism does not allow for the "sharing" of specifications the USES and SEES constructs do, with some limitations. The USES and SEES constructs allow for several machines to have knowledge of another machine, provided that this machine and the seing/using machines are eventually included and instantiated in a single new machine. Neither the USES nor the SEES clauses are transitive, i.e. only the variable, sets and constants of explicitly seen/used machines can be read.

The difference between the two clauses concerns variable scope; variables of a used machine may appear in the invariant of a using machine, but variables of a seen machine may not

 ©1996 *B-Core (UK) Ltd.*

appear in the invariant of the machine that sees it. This restriction means that a seen machine may be refined independently of a machine that sees it. Furhermore, operations of seen machines can be called provided those operations do not change the state (i.e. only "enquiry" operations may be called).

2.5 Abstract Machine Refinement

An Abstract Machine specification is not necessarily executable, and certainly the abstract model of the state of the machine as well as the formulation of the operations may not be directly translatable into executable code (for example, non-determinism plays an important role in many specifications). To facilitate translation into code, the Abstract Machine must be refined.

A refinement can be regarded as a step towards code; it is not the final step (see page 13), but permits a change of state in order to get closer to the data structures of the final implementation - *data refinement* - if required, and also allows for *algorithmic refinement*, and a combination of the two.

Data refinement is achieved by a change of variables and a new set of operations which refine each of those in the original specification. The relationship between the abstract variables and the new variables is given through new invariant conditions. Algorithmic refinement permits the reformulation of operations without changing the state model.

An Abstract Machine specification does not permit sequencing: substitutions are either simple or multiple (parallel). In a refinement, however, sequencing is permitted, the GSL extension being provided by the following law:

GSL Syntax	Equivalent Syntax
$[S \,;\, T]R$	$[S]([T]R)$

The refinement relation defined within the B-Method is based on that presented in [20],[24],[17] and possesses some fundamental properties which are of great importance in performing incremental development and proof. The refinement relation is *transitive*, which means that the final implementation may be verified by verification of each refinement step, The refinement relation is also *monotonic* with respect to all constructs of GSL, which means that subcomponents of an operation can be refined independently.

 ©1996 *B-Core (UK) Ltd.*

The refinement of an Abstract Machine incurs a number of proof obligations. We consider the most general case of simultaneous data and algorithmic refinement.

MACHINE
Machine_name
VARIABLES
x
INVARIANT
P
INITIALISATION
S
OPERATIONS
$z \longleftarrow Op_Name \ =$
PRE
Q
THEN
T
END ;
...
END

REFINEMENT
Refinement_name
REFINES
Machine_name
VARIABLES
y
INVARIANT
R
INITIALISATION
U
OPERATIONS
$z \longleftarrow Op_Name \ =$
PRE
L
THEN
V
END ;
...
END

gives rise to the following proof obligations:

$$\exists(x,y) \cdot (P \ \wedge \ R)$$

$$[U] \ \neg([S] \ \neg R)$$

$$\forall(x,y) \ . \ (\ (P \ \wedge \ R \ \wedge \ Q) \ \ \Rightarrow \text{Ł})$$

$$\forall(x,y) \ . \ (\ (P \ \wedge \ R \ \wedge \ Q) \ \ \Rightarrow \\ (\ [V'] \ \neg([T] \ \neg(R \ \wedge \ z = z'))\ )\ )$$

©1996 *B-Core (UK) Ltd.*

In the above, V' stands for substitution V within which the output variable z has been replaced by z'. The first proof obligation concerns the existence for the new variables of the refinement, and this checks whether the new invariant condition, R, contradicts the invariant condition already given - P; the second deals with the correctness of the initialisation, and it verifies that the new initialisation, U, establishes a situation where the specified initialisation, S, cannot fail to establish the condition - R. (R being the new invariant and the abstraction relationship); the third and fourth ensures the correctness of the operations (there will be a pair of such proof obligation for each operation), and it verifies that an operation, V, establishes a situation where the specified operation, T, cannot fail to maintain R, while preserving output.

2.6 Abstract Machine Implementation

An Abstract Machine implementation represents the last refinement and one which may be directly translated into code; thus an implementation may contain no non-determinism (unlike a refinement), complete concrete algorithms being provided for each operation. Loop constructs are permitted in an implementation and thus GSL is extended by the following syntactic rewriting rule for a while loop:

AMN Substitution	GSL Substitution
WHILE P **DO** S **END**	$(P \Longrightarrow S)\hat{}$ **;** $(\neg P \Longrightarrow \texttt{skip})$

For a description of $\hat{}$, see [2].

Usually a loop is used together with an initialisation (substitution), an invariant (predicate) and, a variant (expression giving a natural number), giving the proof rule enabling us to verify loop constructs without using the definition above:

$$
\begin{array}{l}
\quad [T]Q \ \wedge \\
\quad \forall l \,.\, (\ Q \ \Rightarrow \ E \in \mathbf{N}\ ) \ \wedge \\
\quad \forall l \,.\, (\ (Q \wedge P) \ \Rightarrow \ [S]Q\ ) \ \wedge \\
\quad \forall l \,.\, (\ (Q \wedge P) \ \Rightarrow \ [n := E]([S]E < n\ )) \ \wedge \\
\quad \forall l \,.\, (\ (Q \wedge \neg P) \ \Rightarrow \ R\ ) \\
\Rightarrow \\
\quad [T \ \mathbf{;\ WHILE}\ P\ \mathbf{DO}\ S\ \mathbf{INVARIANT}\ Q\ \mathbf{VARIANT}\ E\ \mathbf{END}]\, R
\end{array}
$$

 ©1996 *B-Core (UK) Ltd.*

The first antecedent expresses the fact that the initialisation, T, establishes the loop invariant - Q; the second that, the loop variant is a natural number expression (note that l denotes the variables used in the loop); the third that, under the guard P, the body, S, maintains the loop invariant; the fourth that, under the guard, the body decreases the loop variant - E (n is a variable not used elsewhere); the last that the post-condition, R, is established when the loop terminates, i.e. when the negation of the guard holds.

The proof obligations of an Abstract Machine implementation are exactly those give for an Abstract Machine refinement on page 12.

2.6.1 Structuring and Implementing Designs

An important difference between an Abstract Machine refinement and an Abstract Machine implementation is that in the latter all of the data is completely encapsulated in other machines; encapsulated variables may not be referenced within the algorithms of an implementation. This is known as the *full-hiding principle*, and means that an implementation need have access only to the specifications of the encapsulating machines; such machines may be independently verified, refined and implemented, so long as the specification is met. This approach has clear and considerable benefits in the construction of large software systems, where teams of designers may work in parallel on different aspects of the system. This also means that Implementations are machine independent description.

Another important consideration in the design of software systems is that of reuse. A library of encapsulating machines is available to the designer, each having an Abstract Machine specification and matching implementation (for example a set machine, sequence machine and so on), to facilitate the process of reaching code from abstract specification quickly, and also to impose standards that are easily-understood and followed.

2.6.2 The IMPORTS and SEES Clauses

The IMPORTS clause implements the full-hiding principle, allowing for the introduction of encapsulating machines into an Abstract Machine implementation. As with the INCLUDES clause for specification construction, it permits machine parameterisation and renaming to permit particular instances as well as distinct copies of machines (including library machines) to be imported.

Since all data is imported, an Abstract Machine implementation does not have a variables clause, but the imported variables may be referenced in the INVARIANT clause (to enable the introduction of "gluing invariants") and also in the invariant part of a loop.

The SEES clause allows for sharing of encapsulating machines which are separately developed. An imported machine can be shared by other machines. These machines SEES the imported machine. While the importing machine has access to all the operations pro-

 ©1996 *B-Core (UK) Ltd.*

vided by the shared machine, the seeing machines have limited access, and cannot use the operations which modify the abstract state of the shared machine.

2.7 The B-Toolkit Components

The B-Toolkit provides powerful fully-integrated tool support for every aspect of the B-Method. These tools are implemented on the B-Platform (the successor to the B-Tool [1], [12], thus providing theorem-proving capabilities). The B-Toolkit and the B-Method underwent seven years of research and development at the PRG in Oxford and at BP Research in Sunbury, and development is now continuing at B-Core (UK) Ltd in Oxford.

2.7.1 The B-Toolkit Managers

Integration of the toolkit is achieved by several cooperating managers; the two most important are the configuration manager, which incorporates a file manager and is responsible for the status of each single construct in a development, and the dependency manager, responsible for maintaining the inter-dependence between constructs.

These managers provide for the introduction (and removal) of constructs within a development, and ensure that when edits are to be committed, the extent of those edits is made known to the developer.

One of their most important functions is to ensure that all parts of the development are up-to-date, and reflect all changes that have been committed to the system (for example, it is impossible to produce code or documents that do not reflect the latest changes made in a specification). Equally important is the facility to automatically remake the system to some previous "high-water" mark once changes have been made (this process may be fine-tuned by the setting of appropriate flags).

Configuration Management is achieved through the Reset tool which, in cooperation with the B-Toolkit Managers, permits a development (at any stage) to be collapsed to its source files (which together with the toolkit dependency and configuration managers enable the complete system to be automatically rebuilt - including proof, documents and code). This facility provides for conformance with emerging standards such as BS 5750 and DEF STAN 00-55.

The B-Toolkit Managers also provide a full range of status and query functions.

2.7.2 Analysis

Processing of Abstract Machine specifications, refinements and implementations begin with the Analyser, providing syntax-checking and the "de-sugaring" of AMN into GSL. The

 ©1996 *B-Core (UK) Ltd.*

former, of course, provides a rich "pseudo-programming" notation, very convenient for humans, the latter providing the concise substitution language which lends itself very well to machine processing and automation. All subsequent processing takes place on the GSL form of the construct.

The Analyser also ensures compliance with the semi-hiding and full-hiding principles.

The TypeChecker will report on the type correctness of each construct; each expression in a construct is assigned a type constructed on the given sets using the two operators $\mathbf{P}$ and $\times$ denoting respectively the power set constructor and the cartesian product constructor. Moreover, in a substitution the types of the elements situated on each side of the $:=$ operator must be the same.

2.7.3 Animation

It is of fundamental importance in any formal development of a computer system that the mathematics "captures" what is actually required; verification can ensure that a specification is internally consistent, or that refinement is correct with respect to its abstraction, but cannot demonstrate compliance with the (usually informal) requirements.

The B-Toolkit Animator provides an important validation facility, often overlooked in formal developments, that enables Abstract Machines to be "executed" and tested against expected behaviour. It may thus may be viewed as a very high-level prototyper (although no design need be provided by the developer).

It is envisaged that as well as a software house building computer systems on the B-Toolkit, the software procurer having the B-Toolkit could take delivery of specifications and refinements as they are written (rather than waiting for the final code to be delivered) and use validation tools such as the Animator to check progress.

2.7.4 Proof

One of the major advantages that a formal development method provides is the facility to provide verification conditions or proof obligations. A criticism of formal methods in the past has been the vast number of proof obligations provided by even the most modest of developments: thus automated machine support in this area is of fundamental importance.

The B-Toolkit POGenerator is an tool that automatically produces all proof obligations associated with Abstract Machine specifications, refinements and implementations. It encodes an efficient calculus implementing the rules presented on pages 6, 8, 9, 11, 12 and 13; a file of proof obligations is produced which may be subsequently presented to the B-Toolkit Provers (or, indeed, external provers if required).

The B-Toolkit provides for automated proof - the Autoprover - and for interactive proof -

 ©1996 *B-Core (UK) Ltd.*

the InterProver. Both are built on a mathematical library containing a rich collection of mathematical laws for the underlying set-theoretic notation. The InterProver provides an environment in which the developer may supplement the library with further rules (such rules should, of course, be verified independently).

Both the POGenerator and AutoProver participate in the automatic Remake facility (if required); user-supplied theory is reused in an attempt to discharge as many of the new proof obligations as possible.

2.7.5 Specification/Module Library

The B-Toolkit Specification/Module Library provides a collection of off-the-shelf Abstract Machine specifications together with matching code, which may be employed in the construction of specifications (using the SEES, USES or INCLUDES facilities) or in the design of implementations (using the IMPORTS or SEES facility); such machines require no further development; in particular, they do not require analysis or proof (since they are guaranteed to be internally consistent) or refinement (since the code is provided).

The use of the library machines facilitate greatly the process of building an abstract specification and of its subsequent implementation.

Library machines provided include those for:

1. Manipulating types of objects: character string, bit string, scalar and boolean.
2. Encapsulation of a single object: variable, array, sequence, partial function and character string.
3. Encapsulation of a collection of objects: sets, sequences, strings and functions.
4. Simple i/o: terminal i/o for basic types of objects (numbers and strings etc), token i/o for entering and displaying elements of a specified type, persistent data facilities.

Each library machine provides a rich collection of operations for manipulating and accessing the encapsulated data.

A facility is also provided for the establishment of team libraries, so that specification/module pairs already developed are made accessible to other developers.

All library machines may be submitted to the Animator and to the automatic Interface-Generator (see below).

 ©1996 *B-Core (UK) Ltd.*

2.7.6 Code and Interface Generation

The B-Toolkit CodeGenerator translates appropriate Abstract Machine implementations into object code; library modules participate fully in this process. When a development is complete, the Linker will produce executable code.

The InterfaceGenerator will automatically provide an interface with executable code for any Abstract Machine specification provided that it, and all subordinate constructs, have implementations built on the toolkit library; the i/o library machines will be used to provide appropriate functionality for each operation parameter.

The InterfaceGenerator thus provides a rapid-prototyping facility.

The CodeGenerator, Linker and InterfaceGenerator each participate in the automatic Remake facility (if required).

2.7.7 Enumerator and Base Generators

The Enumerator generates the access functions associated with a given enumerated set; and Abstract Machine specification and implementation (built on the library i/o machines) are generated.

The Base Generator is a powerful tool which interprets a declarative description of a database requirement (a syntactic form of entity relationship modelling) and automatically generates an Abstract Machine specification modelling that requirement, together with a rich set of operations to manipulate and access the data; an Abstract Machine implementation is also automatically generated, and built on the library object machines.

All such generated Abstract Machines may be incorporated in the construction and design of other Abstract Machine components; they may be submitted to the Animator and to the InterfaceGenerator. The Enumerator and Base generator participate in the automatic Remake facility (if required).

2.7.8 Documentation

All Abstract Machine components (including interfaces, enumerations, bases and so on) may be submitted to the DocumentMarkUp tool which will automatically produce and compile LaTeX source code. Informal annotations may be included (which will, of course, be ignored by all other tools).

Components may be gathered together into a single Document, which provides full cross-referencing of variable, set, constant, operation and construct names at the clause and construct level, together with an index for the entire document. These features are automatic, but may be turned off.

 ©1996 *B-Core (UK) Ltd.*

As with all other tools, the DocumentMarkUp tool participates in the automatic Remake facility, if required.

2.8 Conclusion

The B-Method is designed to provide a homogeneous language, methodology and integrated tool support for the formal specification, design and implementation of real-life software systems. Therefore, the features of incremental construction and proof have been guiding principles in its development. It has now reached a mature and stable form, and a full account of the B-Method and its theoretical foundations is to appear as a book by J-R. Abrial [2].

The B-Toolkit has been under alpha- and beta-test since January 1992, and was released in June 1994; The B-Method has been in use at GEC-Alsthom, which has for some years used a special tool environment to specify and develop railway control systems [14] [13],[15]. A prototype version of the B-Toolkit was used at BP International in building a fully operational graphics system [11] for visualising the flow of oil inside a refinery. MATRA Transport is using the current version of the B-Toolkit for software specification, and IBM (UK) is using it in the complete development of a new sub-system within CICS/ESA. The Parisian subway authority, RATP, and the French railways authorities, SNCF and INRETS, are funding the introduction of the B-Method into the French transport industries.

 ©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

B-TOOLKIT USERS' MANUAL

SECTION 3

Abstract Machine Notation (AMN)

©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

3 Abstract Machine Notation (AMN)

3.1 Abstract Machine Notation

3.1.1 Overview

This description of the Abstract Machine Notation is organised as follows:-

- AMN Substitutions
- AMN Operations
- Machines
- Refinements
- Implementations

Some constructs require the use of a type-determining predicate to be well-formed. A type-determining predicate `P` must contain a constraining predicate of the form `xx: S`, `xx < S`, `xx <: S` or `xx = E`, where `xx\S`, `xx\E`.

3.1.2 AMN Substitutions

The generic syntactic variables are as follows: P, Q, R for predicates; S, T, U, V for AMN substitutions, E,F for expressions; ff, xx, yy for variables; l, m, n for lists of expressions; vlx for variable lists; opn for AMN operation names; asub for AMN substitutions. z\E indicates that there are no free occurrences in E of the variables in z. The definitions below are given in terms of Generalised Substitutions. The syntax is presented informally, and occurrences of '...' mean that any number of the surrounding construct can occur.

```
BEGIN S END             =   S

PRE P THEN S END        =   P | S

IF P THEN S ELSE T END =   (P ==> S) [] (not(P) ==> T)

IF P THEN S END         =   IF P THEN S ELSE skip END


IF P THEN               =   IF P THEN
  S                             S
```

 ©1996 *B-Core (UK) Ltd.*

```
ELSIF Q THEN
  T
...
ELSIF R THEN
  U
ELSE
  V
END
```

```
ELSE
  IF Q THEN
    T
  ...
    ELSE
       IF R THEN
         U
       ELSE
         V
       END
    END
  ...
END
```

```
IF P THEN
  S
ELSIF Q THEN
  T
...
ELSIF R THEN
  U
END
```

=

```
IF P THEN
  S
ELSIF Q THEN
  T
...
ELSIF R THEN
  U
ELSE
  skip
END
```

```
CHOICE
  S
OR
  T
OR
 ...
OR
  U
END
```

=

```
S [] T [] ... U
```

```
SELECT P THEN
  S
WHEN Q THEN
  T
...
```

=

```
P ==> S []
Q ==> T []
... []
R ==> U
```


©1996 *B-Core (UK) Ltd.*

```
WHEN R THEN
  U
END
```

```
SELECT P THEN          =   SELECT P THEN
  S                            S
WHEN Q THEN                WHEN Q THEN
  T                            T
...                        ...
WHEN R THEN                WHEN R THEN
  U                            U
ELSE                       WHEN not(P) & not(Q) & ... & not(R) THEN
  V                            V
END                        END
```

```
CASE E OF              =   SELECT E: {l} THEN
  EITHER l THEN              S
    S                      WHEN E: {m} THEN
  OR m THEN                  T
    T                      ...
  ...                      WHEN E: {n} THEN
  OR n THEN                  U
    U                      ELSE
  ELSE                       V
    V                      END
  END
END
```

```
CASE E OF               =  SELECT E: {l} THEN
 EITHER l THEN                S
   S                        WHEN E: {m} THEN
 OR m THEN                    T
   T                        ...
 ...                        WHEN E: {n} THEN
 OR n THEN                    U
   U                        ELSE
 END                          skip
END                         END

VAR xx IN               =  @xx.S
```

 ©1996 *B-Core (UK) Ltd.*

```
  S
END

ANY xx WHERE P          =  @xx.(P ==> S). P must contain a
THEN                        type determining predicate.
  S
END

LET xx BE               =  @xx.(xx = E ==> S), where xx\E.
  xx = E
IN
  S
END
```

`S || T`		Parallel substitution (S and T are substitutions on disjoint sets of variables.
`xx:= E || yy:= F`	=	`xx,yy:= E,F`
`opn`		An AMN substitution, with name opn.
`opn(vl1)`		A parameterised AMN substitution, with input vl1.
`vl2 <-- opn`		A parameterised AMN substitution, with output vl2.
`vl2 <-- opn(vl1)`		A parameterised AMN substitution, with input vl1 and output vl2.

```
xx:= bool(P)            =  IF P THEN
                             xx:= TRUE
                           ELSE
                             xx:= FALSE
                           END
```

`skip`		No-op

```
xx:: E                   =  @xx'(xx': E ==> xx:= xx')

xx: P                    =  @xx'.([xx:= xx']P ==> xx:= xx'). P must contain a
                            type determining predicate.
```

 ©1996 *B-Core (UK) Ltd.*

```
ff(xx):= E              =  ff:= ff <+ {xx|->E}

S;T                        Identical to its Generalised Substitution
                           counterpart, S;T.


S;WHILE P DO            =  S, followed by looping on T so long as the guard P
    T                      holds, with loop variant E and loop invariant Q.
  INVARIANT
    Q
  VARIANT
    E
  END
```

3.1.3 AMN Operations

The generic variables are as follows: `vlx` for variable lists; `opn` for operation names; `asub` for AMN substitutions. AMN operations have the following forms:

```
opn = asub

opn(vl1) = asub

vl2 <-- opn = asub

vl2 <-- opn(vl1) = asub
```

3.1.4 Abstract Machines

Abstract Machine constructs are described by the following BNF-like syntax; the order in which optional clauses appear is not significant.

Here, the syntax
`exp1 | exp2` indicates `exp1` or `exp2` (choice),
`< exp >` indicates zero or one occurrence of `exp` (optionality) and
`<< exp >>` indicates zero or more occurrences of `exp` (repetition).

Identifier, UpperCaseIdentifier, Rule, Bnumber, Formula and ProgramLikeFormula are as defined in the section on the B-Platform (section 6) and in the B-Tool Reference Manual.

 ©1996 *B-Core (UK) Ltd.*

An informal description is given, explaining the usages of the clauses of the syntax. In the following, 'become known' means that the identifier in question is in scope within the basic clauses (SETS , PROPERTIES, ASSERTIONS, INVARIANT, INITIALISATION, OPERATIONS, unless otherwise stated.

3.1.5 Machines

```
MACHINE              Identifier  < ( param_list ) >

                     Introduces an optionally Parameterised machine.
                     A parameter is an Identifier denoting a set or an
                     element of a set.
                     Upper Case Identifiers (e.g. SS) denote non-empty set.
                     Any other parameter (pp) is either a Natural Number, or,
                     if a constraining predicate (e.g. pp : SS) is given in
                     the CONSTRAINTS clause,  an element of one of the
                     parameter sets.

< CONSTRAINTS        Formula  << &   Formula  >> >

                     A list of predicates separated by ( & ) constraining
                     the parameters of the machine.

< USES               rnm_machine_ref << ,  rnm_machine_ref  >> >

                     USES can be used to get access to machines which
                     will be included elsewhere. Normally, a used machine
                     and the using machine will be included in the same
                     INCLUDES clause in another machine. The USES clause
                     is a list of machines.
                     The variables, sets and constants of a used machine
                     'become known' to the using machine; they can appear
                     in the invariant, and can appear in AMN Substitutions
                     provided that they are not modified. Renamed machines
                     (e.g. ver_A.mch_B) can be used. A machine can be used
                     (shared) by several other machines.  Parameterised
                     machines can be used, but no reference can be made to
                     their formal parameters, and ONLY the the machine
                     which include a used machine can provide actual
                     parameters.
                     SEES can be used in stead of USES if the seen variables
                     does not need to be referred to in the invariant.
```

 ©1996 *B-Core (UK) Ltd.*

< SEES rnm_machine_ref << , rnm_machine_ref >> >

SEES can be used to get access to machines which
will be included or imported elsewhere. Normally,
a seen machine and the seeing machine will be
imported in the same IMPORTS clause in an
implementation, and both machines will be refined and
implemented separately.
The SEES clause is a list of machines seen by this
machine. The variables, sets and constants "become known&quot
to the seeing machine. However, the variables cannot
appear in the invariant, and can only appear in AMN
Substitutions provided that they are not modified.
Renamed machines (e.g. ver_A.mch_A) can be seen.
A machine can be seen (e.g. shared) by several other
machines. Parameterised machines can be seen, but no
reference can be made to their formal parameters, and
ONLY the construct which include (or import) a seen
machine can provide actual parameters.

< INCLUDES rnm_machine_ref < (param_list) >
<< , rnm_machine_ref < (param_list) > >> >

INCLUDES is the main facility for constructing
specifications.
The clause is a list of possibly instantiated
machines which all become part of this present
machine.
The sets and constants of the included machine
become sets and constants of the including machine.
The variables of all included machines become
variables of the including machine, and can appear
in the invariant. All the variables of included and
including machines must be disjoint.
Note that, an included variable can only be modified
indirectly through the use of operations defined
in the machine which introduced the variable.
Included machines can be renamed, for example
ver_A.mch_B. As a consequence of this renaming, all
included variables and operations can only be
referred to by prefixing their names with ‘ver_A.‘.
Renaming can be used to include multiple copies of a

 ©1996 *B-Core (UK) Ltd.*

single machine. Actual parameters of included machines can contain references to sets and constants of other included machines.

< PROMOTES rnm_operation_ref << , rnm_operation_ref >> >

A list of operations from included machines. These operations become operations of the including machine.

< EXTENDS rnm_machine_ref < (param_list) >
<< , rnm_machine_ref < (param_list) > >> >

A list of machines that the present machine extends. The machines are included and all operations of these machines are promoted. Extended machines can be renamed, for example ver_A.mch_B.

< SETS UpperCaseIdentifier | enumerated_set
<< ; UpperCaseIdentifier | ; enumerated_set >> >

< CONSTANTS Identifier << , Identifier >> >

A list of constants. The constants ‘MaxScalar‘ and the set ‘SCALAR‘ where MaxScalar = 2147483646 and SCALAR = 0..MaxScalar are defined in a library machine (Scalar_TYPE) available within the B-Toolkit Library.

< PROPERTIES Formula << & Formula >> >

A list of predicates separated by (&) which gives the properties of the constants. This list must contain a constraining predicate for each constant.

< VARIABLES Identifier << , Identifier >> >

A list of state variables.

< INVARIANT Formula << & Formula >> >

A list of predicates separated by ‘&‘ constraining the values the variables of the machine can take. This list must contain a constraining predicate for each variable.

 ©1996 *B-Core (UK) Ltd.*

```
< ASSERTIONS        Formula << & Formula >> >

                    A list of of predicates separated by ‘&‘ which gives
                    properties which can be asserted from the machine
                    invariant and other contextual information.


< DEFINITIONS       Formula == Formula << ; Formula == Formula >>  >

                    A list of definitions used in this machine. Each
                    definition is of the form l == r, where ‘l‘ and ‘r‘
                    are B formulae, e.g
                    name(a,b) == &quot;formula in which ’a’ and ’b’ are jokers&quot;


< INITIALISATION    Formula >

                    A substitution which establishes the invariant.
                    All machine variables must be set. Included (and seen
                    and used) variables are initialised automatically.

< OPERATIONS        operation  << ; operation >> >

                    A list of operations for the machine. Sequencing ‘;‘ and loop
                    (e.g. WHILE) substitutions cannot be used.

END
```

.

```
rnm_machine_ref   ::=  < rename_prefix . > Identifier


rnm_operation_ref ::=  < rename_prefix . > Identifier

rename_prefix     ::=  Identifier
```

 ©1996 *B-Core (UK) Ltd.*

3.1.6 Refinements

The first clause indicates which machine is being refined. Note that any AMN Substitution can be used in the operation list. New variables may be introduced in a refinement. If a variable is giving a name identical to a name of a variable in the machine being refined ,then we assume that no data-refinement has taken place on that variable - i.e. the two variables are identical.

```
REFINEMENT          Identifier

                    Introduces a  refinement.  This  refinement inherits
                    the parameters, sets, constants, constraints and
                    properties  of the machine it refines.

REFINES              machine_ref

                    The name of the machine (abstraction) being refined.
                    All  variables of the machine being refined become
                    known to this refinement.  The variables of the
                    abstraction can appear in the invariant of the
                    refinement (that defines the abstraction relationship).
                    However, they cannot be read or modified within an AMN
                    substitution of the refinement.


< SEES              machine_ref  << , machine_ref >> >

                    A list of machines seen by this refinement. The
                    variables, sets and constants become known to the seeing
                    machine.  However, the variables cannot appear in the
                    invariant, and can only appear in AMN Substitutions
                    provided that they are not modified. Seeing is
                    used for sharing and machines renamed elsewhere
                    (e.g. ver_A.mch_A) in an IMPORT clause can be seen.

< SETS              UpperCaseIdentifier | enumerated_set
                    << ; UpperCaseIdentifier | ; enumerated_set >> >

< CONSTANTS         Identifier << , Identifier >> >

< PROPERTIES        Formula << & Formula >> >
```

 ©1996 *B-Core (UK) Ltd.*

A list of predicates separated by & which gives the properties of and constants. This list must contain a constraining predicate for each new constant.

< VARIABLES Identifier << , Identifier >> >

A list of all concrete state variables of the refinement. If a concrete state variable has the same name as a variable of the abstraction (i.e. the construct being refined) then the abstract variable can only be referred to by appending ‘dollar‘ 0 to the name. Furthermore, the value of the concrete variable is assumed to be equal to the similar named variable from the abstraction. This facility is used for algorithmic refinements.

< INVARIANT Formula << & Formula >> >

A list of predicates separated by & constraining the values the variables can take and defining the relation between the concrete and abstract state variables. This list must contain a constraining predicate for each new variable of the refinement.

< ASSERTIONS Formula << & Formula >> >

A list of predicates separated by ‘&‘ which gives conditions that can asserted from the invariant condition and the available contextual information.

< DEFINITIONS Formula == Formula << ; Formula == Formula >>

A list of definitions used in this refinement. Each definition is of the form l == r, where ‘l‘ and ‘r‘ are B formulae.

< INITIALISATION Formula >

A substitution which initialises the concrete state variables consistently with the change of variable. All new variables must be initialised.

 ©1996 *B-Core (UK) Ltd.*

```
< OPERATIONS        operation << ; operation >> >

                    A list of operations refined in this definition.


END
```

3.1.7 Implementations

An implementation is the final refinement of a machine. In an implementation, all operations are ready to be translated into code, and imported operations can be invoked from these those operations.

```
IMPLEMENTATION      Identifier

                    Introduces an implementation.  The implementation
                    inherits the parameters, sets, constants, constraints
                    and properties of the machine it implements.

REFINES             machine_ref

                    The name of the machine being implemented.  All
                    variables of the machine being implemented (the
                    abstraction) become known to this implementation.
                    The variables of the abstraction can appear in the
                    state-invariant (and loop-invariants) of the implementation
                    (that defines the abstraction relationship).  However,
                    they cannot be read or modified within an AMN
                    substitution of the implementation.

< SEES              machine_ref  << , machine_ref >> >

                    A list of machines seen by this implementation.
                    The variables, sets and constants become known to the
                    seeing machine.  However, the variables cannot appear
                    in the invariant. Seen constants can be referred to
                    in AMN Substitutions, but  variables and sets from
                    the seen machine cannot appear in AMN Substitutions.
                    Operations from the seen machine can be used,
                    provided that they do not modify the seen variables
                    (e.g. inquiry operations).  Any seen constant which
```

 ©1996 *B-Core (UK) Ltd.*

is actually used must be an element of a deferred set or a parameter. Machines, renamed elsewhere (e.g. ver_A.mch_B) can be seen. A seen (shared) machine must therefore be imported somewhere else.

< IMPORTS

machine_ref < (act_param_list) >
<< , machine_ref < (act_param_list) > >> >

A list of (possibly instantiated) imported machines. The variables, sets and constants of these machines become known to the importing implementation, and they can appear in the invariant of the importing implementation. However they cannot appear in the operations. All operations of imported machines can be used by the importing implementation. The actual parameters must be provided.

< PROMOTES

operation_ref << , operation_ref >> >

A list of operations from imported machines. Promoted operations become operations of the importing machine.

< SETS

UpperCaseIdentifier | enumerated_set
<< ; UpperCaseIdentifier | ; enumerated_set >> >

< CONSTANTS

Identifier << , Identifier >> >

A list of constants, all of which must be defined in the PROPERTIES clause.

< PROPERTIES

Formula << & Formula >> >

A list of predicates separated by &, which defines constants and sets. Each deferred set is given a VALUE. This will normally be an existing enumerated set, an interval or a deferred set from an imported machine.

< INVARIANT

Formula << & Formula >> >

A list of predicates which defines the relationship between the concrete (imported) and abstract state variables.

©1996 *B-Core (UK) Ltd.*

```
< ASSERTIONS        Formula << & Formula >> >

                    A list of assertions  implied by the invariant and
                    other contextual information.

< DEFINITIONS       Formula == Formula << ; Formula == Formula >> >

< INITIALISATION    Formula >

                    A substitution which by using the imported operations
                    modifies the variables of the imported machines.
                    Note, that the imported machine variables are set
                    according to their own initialisations clause before
                    this modification.

< OPERATIONS        operation << ; operation >> >

                    A list of operations. Non-deterministic substitutions
                    (CHOICE, SELECT, ANY, LET , ::, :) cannot be used.
                    The simultaneous substitution operator || and
                    preconditioned substitutions cannot be used.  The
                    imported variables can only be tested and modified by
                    using the imported operation.
END


param_list     ::=  Identifier << , Identifier >>

act_param_list ::=  act_param << , act_param >>

act_param      ::=  Bnumber | Identifier | Formula |
                     Bnumber .. Bnumber | { set_contents }

enumerated_set ::=  UpperCaseIdentifier = { set_contents }

set_contents   ::=  Identifier << , Identifier >> >

machine_ref    ::=  Identifier

operation_ref  ::=  Identifier

operation      ::=  <  Param_list  <-- > Identifier < (  param_list  ) >  =
                     Substitution
```

 ©1996 *B-Core (UK) Ltd.*

A full on-line help listing is available in the Contents Page
See also the complete Index. ©**B-Core (UK) Limited**, Last updated:

©1996 *B-Core (UK) Ltd.*

3.2 Mathematical Notation

See the 'B-Method Abstract Machine Notation Summary' for a full treatment of the mathematical notation.

3.2.1 Contents

- Predicates
- Expressions
- Sets
- Natural_Numbers
- Relations
- Functions
- Sequences
- Generalised Substitutions
- Variables, Variable Lists and Identifiers
- Infix Operator Priorities

3.2.2 Notes on operator binding

Compound formulae (e.g. `A => B & C`) are given an unambiguous interpretation by the operator binding rules:

1. All operators bind to the left (are left-associative) except "." which binds to the right.
2. Each symbol (e.g. `&`) is given a priority, and the highest priorities bind strongest, e.g. `A => B & C` is equivalent to `A => (B & C)`.
3. In case of equal priority the leftmost operator binds the strongest, e.g. `A & B & C <=> (A & B) & C`.

The priorities of infix operators are listed in the table below.

 ©1996 *B-Core (UK) Ltd.*

3.2.3 Infix Operator Priorities

Priority	Operator
10	.
3	mod * /
2	- +
1	..
0	/\ \/ |->
0	<| <<| |> |>> <+ +> >< circ
0	^ -> <- /| |/
0	< <= > >= /= /:
-1	<-> --> +->
-1	>-> >+> +->> >->> -->>
-1	<-- ,
-2	<: <<: /<: /<<:
-2	:=
-4	= == : <=> ::
-5	& or
-6	=> ==>
-7	; || []
-8	|

A full list of priorities is found in the B Toolkit Symbol Table, $BKIT/BLIB/AMNSYMBOL.

3.2.4 Predicates

Let `z` be a Variable List, `x` Variable, `E` and `F` be Expression Lists, `P` and `Q` be Predicates, and `S`, `T` be Sets. `z\E` means that there are no free occurrences in `E` of the variables in `z`.

3.2.5 General Predicates

`P & Q`
: Conjunction: "`P` and `Q`".

`P => Q`
: Implication: "`P` implies `Q`" or "if `P` then `Q`".

`not(P)`
: Negation: "Not `P`".

 ©1996 *B-Core (UK) Ltd.*

`!z.(Q => P)`
: Universal quantification: "For all z where Q, P". The predicate Q must, for each variable x in the list z, contain a constraining predicate, i.e. `x: S`, `x <: S`, `x <<: S` or `x = E`, where `z\S, z\E`.

`P or Q`
: Disjunction: "`P` or `Q`".

`P <=> Q`
: Equivalence: "`P` is equivalent to `Q`". An abbreviation for `(P => Q) & (Q => P)`.

`#z.P`
: Existential quantification: "For some `z`, `P` holds". The predicate `Q` must, for each variable `x` in the list `z`, contain a constraining predicate, i.e. `x: S`, `x <: S`, `x <<: S` or `x = E`, where `z\S, z\E`.

3.2.6 Predicates on Expressions

`E = F`
: Equality: `E` equals `F`.

`E /= F`
: Inequality: `E` is not equal to `F`.

3.2.7 Expressions

Let E and F be Expressions.

`E,F`
: Expression list.

`E |-> F`
: Ordered pair (maplet).

3.2.8 Sets

Let `z` be a Variable List, `P` be a Predicate, `E` and `F` be Expressions, and `S` and `T` be sets.

`E : S`
: Set membership: the predicate "`E` belongs to `S`" or "`E` is an element of `S`".

`E /: S`
: Set non-membership: the predicate "`E` does not belong to `S`", i.e. `not(E: S)`.

 ©1996 *B-Core (UK) Ltd.*

`S <: T`
Set inclusion: the predicate "`S` is included in `T`", i.e. "every element of `S` is also an element of `T`".

`S /<: T`
Set non-inclusion: the negation of the predicate `S <: T`.

`S <<: T`
Strict set inclusion: the predicate "`S` is included in `T`, but is not equal to `T`".

`S /<<: T`
String set non-inclusion: the negation of the predicate `S <<: T`.

3.2.9 Set Expressions

`{z | P}`
Set comprehension: the subset such that `P`. The predicate `P` must, for each variable `x` in the list `z`, contain a constraining predicate, i.e. `x: S`, `x <: S`, `x <<: S` or `x = E`, where `z\S`, `z\E`.

`{z | z: S & P}`
Set comprehension: the subset of `S` such that `P`.
e.g. `{x,y | x,y: S*T & P}`.

`S * T`
Cartesian product: the set of Ordered Pairs whose first component is from `S` and second component is from `T`.

`POW(S)`
Power set: set of all subsets of `S`.
`x: POW(S) <=> x <: S`.

`S \/ T`
Set union: the set of elements which are elements of `S` or `T`.

`S /\ T`
Set intersection: the set of elements which are elements of `S` and `T`.

`S - T`
Set difference: the set of elements which are elements of `S`, but not of `T`.

`{}`
Empty set: the set with no elements.

 ©1996 *B-Core (UK) Ltd.*

`POW1(S)`
Non-empty subset: Set of all non-empty subsets of `S`.
`POW1(S) = POW(S) - {}`.

`FIN(S)`
Finite subsets: Set of all finite subsets of `S`.

`FIN1(S)`
Non-empty finite subsets: Set of all non-empty finite subsets of `S`.
`FIN1(S) = FIN(S) - {}`.

`{E}`
Singleton set: Provided that `E` is not an Expression List, and `E: S`, `{E}` is a singleton set: `{x | x: S & x = E}`.

`{E,F}`
Set enumeration: Provided that `F` is not an Expression List, this is the set with elements from `{E}` together with element `F`. `{E,F} = {E} \/ {F}`.
Note that `F` is an element of `{(E,F)}`, while `E |-> F` is the single element of `{E |-> F}`.

`union(U)`
Generalised union: the generalised union of a set `U` of subsets of `S` `(U: POW(POW(S)))`.
`union(U) = {x | x: S & #s.(s: U & x: s)}`.

`inter(U)`
Generalised intersection: the generalised intersection of a set `U` of subsets of `S` `(U: POW(POW(S)))`. `inter(U) = {x | x: S & !s.(s: U => x: s)}`.

`UNION(z).(P | E)`
Generalised union of the sets `E` where `z` satisfies `P`. For each variable `x` in the list `z`, `P` must contain a constraining predicate of the form `x: S`, `x <: S`, `x <<: S` or `x = F` with `z\s`, `z\F`.
`!z.(P => E <: T) => UNION(z).(P | E) = {x | x: T & !z.(P => x: E)}`.

`INTER(z).(P | E)`
Generalised intersection of the sets `E` where `z` satisfies `P`. For each variable `x` in the list `z`, `P` must contain a constraining predicate of the form `x: S`, `x <: S`, `x <<: S` or `x = F` with `z\s`, `z\F`.
`!z.(P => E <: T) => INTER(z).(P | E) = {x | x: T & #z.(P & x: E)}`.

3.2.10 Natural Numbers

A Natural Number (i.e. a non-negative integer) is an Expression, and the Natural Numbers form an infinite set. Let `m` and `n` be Natural Numbers, `E` and `F` be Expressions, and `P` be a Predicate.

 ©1996 *B-Core (UK) Ltd.*

3.2.11 Predicates on Natural Numbers

`m > n`
: Strict inequality: `m` is greater than `n`.

`m < n`
: Strict inequality: `m` is less than `n`.

`m >= n`
: Inequality: `m` is greater than or equal to `n`.

`m <= n`
: Inequality: `m` is less than or equal to `n`.

3.2.12 Natural Number Expressions

`NAT`
: The set of natural numbers.

`NAT1`
: The set of non-zero natural numbers.

`min(S)`
: Minimum of a non-empty subset, `S`, of `NAT`.

`max(S)`
: Maximum of a non-empty finite subset, `S`, of `NAT`.

`m+n`
: Addition: the sum of `m` and `n`.

`m-n`
: Difference: the difference of `m` and `n` (defined for `m >= n`).

`m*n`
: Product: the product of `m` and `n`.

`m/n`
: Division: the integer division of `m` by `n`.

`m mod n`
: Remainder: the remainder of the integer division of `m` by `n`.

`n .. m`
: Interval: the set of non-negative integers between `n` and `m` inclusive.

 ©1996 *B-Core (UK) Ltd.*

`card(S)`
Cardinality: the cardinality of the finite set `S`: the number of elements in `S`.

`SIGMA(z).(P | E)`
Set summation: the sum of values of the natural number expression `E`, for `z` such that `P` holds. For each variable `x` in the list `z`, `P` must contain a constraining predicate of the form `x: S`, `x <: S`, `x <<: S` or `x= F`, where `z\S`, `z\F`.

`PI(z).(P | E)`
Set product: the product of values of the natural number expression `E`, for `z` such that `P` holds. For each variable `x` in the list `z`, `P` must contain a constraining predicate of the form `x: S`, `x <: S`, `x <<: S` or `x= F`, where `z\S`, `z\F`.

3.2.13 Relations

A Relation is a set of Ordered Pairs. Therefore, any set operation may also be applied to Relations. Let `S`, `T`, `U` and `V` be sets, and `r`, `r1`, `r2` be relations from `S` to `T`, and let `E` and `F` be Expressions. Also let `s <: S` and `t <: T`.

3.2.14 Relational Expressions

`S <-> T`
Relation: Set of relations from `S` to `T`. Equivalent to `POW(S * T)`.

`dom(r)`
Domain of `r`:
The set `{x | x: S & #y.(x,y: r)}`.

`ran(r)`
Range of `r`:
The set `{y | y: T & #x.(x,y: r)}`.

`p;q`
Relational composition: Composition of relations `p` and `q`, where `p: S <-> T` and `q: T <-> U`.
The set `{x,z | x,z: S * U & #y.(y: T & x,y: p & y,z: q)}`. Also denoted by `q circ p`.

`q circ p`
Composition of relations `q` and `p`. The same as `p;q`.

`id(S)`
Identity on `S`.
The set `{x,y | x,y: S * S & x = y}`.

 ©1996 *B-Core (UK) Ltd.*

`s <| r`
Restriction of `r` by `s`. Also known as domain restriction. The relation formed from `r` by keeping only the pairs where the first element is in `s`.
The set `{x,y | x,y:  r & x: s}`.

`r |> t`
Co-restriction of `r` by `t`. Also known as range restriction. The relation formed from `r` by keeping only those pairs where the last element is in `t`.
The set `{x,y |x,y: r & y: t}`.

`s <<| r`
Anti-restriction of `r` by `s`. Also known as domain subtraction. The relation formed from `r` by keeping only those pairs where the first element is in the complement of `s`.
The set `{x,y | x,y: r & x: S-s}`.

`r |>> t`
Anti-co-restriction of `r` by `t`. Also known as range subtraction. The relation formed from `r` by keeping only those pairs where the last element is in the complement of `t`.
The set `{x,y | x,y: r & y: T-t}`.

`r~`
Inverse of `r`. The relation formed from `r` by interchanging the elements of each pair.
The set `{y,x | y,x: T * S & x,y: r}`.

`r[s]`
Image of set `s` under relation `r`.
The set consisting of all those elements related to some element in the set `s` through relation `r`.
The set `{y | y: T & #x.(x: s & x,y: r)}`

`r1 <+ r2`
Overriding of `r1` by `r2`.
The set `(dom(r2) <<| r1) \/ r2`.

`r1 +> r2`
Overriding of `r2` by `r1`.
The set `r2 <+ r1`.

`p >< q`
Direct product of `p` and `q`, where `p: S <-> U` and `q: S <-> V`.
The set `{x,(y,z) | x,(y,z): S * (U * V) & x,y: p & x,z: q}`.

`p || q`
Parallel product of `p` and `q`. where `p: S <-> T` and `q: V <-> U`.
The set `{(x,y),(m,n) | (x,y),(m,n): (S*T) * (V*U) & (x,m: p & y,n: q)}`.

 ©1996 *B-Core (UK) Ltd.*

`iterate(r,n)`
The nth iterate of `r` (where `n: NAT`), i.e. `r` composed with itself `n` times (defined only for `r: S <-> S`).
`iterate(r,0) = id(S)` and `iterate(r,n+1) = r;iterate(r,n)`.

`closure(r)`
The reflexive transitive closure of `r` (defined only for `r: S <-> S`).
`closure(r) = UNION(n).(n: NAT | iterate(r,n))`.

`prj1(S,T)`
Projection: `prj1(S,T) = {(x,y),z | (x,y),z: (S*T)*S & z = x}`.

`prj2(S,T)`
Projection: `prj2(S,T) = {(x,y),z | (x,y),z: (S*T)*T & z = y}`.

3.2.15 Functions

A Function is a Relation with the additional property that each element of the domain is related to a unique element in the range. Any operation applicable to Relations may also be applied to Functions. Let `S` and `T` be sets, `z` a Variable List, `E` be an Expression, and `P` be a predicate.

`S +-> T`
Set of partial functions from `S` to `T` (also known as 'many-to-one relations').
The set `{r | r : S <-> T & (r~;r) <: id(T)}`.

`S --> T`
Set of total functions from `S` to `T`.
The set `{f | f : S +-> T & dom(f) = S}`.

`S >+> T`
Set of partial injections from `S` to `T` (also known as 'one-to-one relations').
The set `{f | f : S +-> T & f~ : T +-> S}`.

`S >-> T`
Set of total injections from `S` to `T`.
The set `S >+> T /\ S --> T`.

`S +->> T`
Set of partial surjections from `S` to `T`.
The set `{f | f: S +-> T & ran(f)=T}`.

`S -->> T`
Set of total surjections from `S` to `T`.
The set `S +->> T /\ S --> T`.

 ©1996 *B-Core (UK) Ltd.*

`S >->> T`
Set of bijections from `S` to `T`.
The set `S -->> T /\ S >-> T`.

`%z.(z: S & P | E)`
Function construction. The function `{x,y | z: S & y=E & P}` where `y\E` and `y\P`, with domain `{z | z: S & P}`.

`%z.(P | E)`
Function construction. The predicate `P` must, for each variable `x` in the list `z`, contain a constraining predicate i.e. `x: S`, `x <: S`, `x <<: S` or `x = E`, with `z\S`, `z\E`.

`f(x)`
For `x: dom(f)`, `f(x)` denotes the value of the function `f` at `x`, i.e. `x |-> f(x): f`.

3.2.16 Sequences

A sequence over a set `S` is a function from `NAT` to `S` whose domain is an interval `1..n` for some natural number `n`. Let `s`, `t` be sequences of elements from `S`, `e` be an element of `S`, and `E` and `F` be expressions.

`<>`
The empty sequence.

`seq(S)`
The set of finite sequences of elements from `S`.

`seq1(S)`
The set of finite non-empty sequences of elements from `S`. `seq1(s) = seq(s) - {<>}`.

`iseq(S)`
The set of injective sequences of elements from `S`. `iseq(S) = seq(S) /\ (NAT1 >+> S)`.

`perm(S)`
The set of bijective sequences of elements from a finite set `S`. A sequence belonging to `perm(S)` is said to be a 'permutation' of `S`. For finite `S`, `perm(S) = 1..card(S) >->> S`.

`s^t`
The concatenation of sequences `s` and `t`.

`e -> s`
The sequence formed by prepending `e` to `s`.

`s <- e`
The sequence formed by appending `e` to `s`.

 ©1996 *B-Core (UK) Ltd.*

`[E]`
Provided that `E` is not an Expression List, `[E]` is the singleton sequence with element `E`, i.e `[E] = E -> <>`.

`[E,F]`
Provided `F` is not an Expression List, then this is `[E]` with `F` appended. Equivalent to `[E] <- F`.

`size(s)`
The size of the finite sequence `s`.

`rev(s)`
The reverse of `s`.

`s /|\ n`
The sequence obtained from `s` by retaining only its first `n` elements, where `n <= size(s)`.

`s \|/ n`
The sequence obtained by removing the first `n` elements of `s`, where `n <= size(s)`.

`first(s)`
The first element of the non-empty sequence `s`.

`last(s)`
The last element of the non-empty sequence `s`.

`tail(s)`
The sequence `s` with its first element removed (`s` must be non-empty).

`front(s)`
The sequence `s` with its last element removed (`s` must be non-empty).

`conc(s)`
The generalised concatenation of a sequence of sequences, `s`. For a sequence `t`, `conc(<>) = <>` and `conc(s <- t) = conc(s)^t`.

3.2.17 Variables, Variable Lists and Identifiers

A Variable is an Identifier. An Identifier is a string of length 2 or more of alphanumeric characters (a to z, A to Z, 0 to 9 ASCII codes) or underscore '_', with at least one letter. An Upper Case Identifier is an Identifier made only from upper case letters and underscore. An Infix operator is either a string of non-alphanumeric characters (excluding '_' "' '$' and '?') or an Identifier declared as an Infix Operator in the AMN Symbol Table, e.g. 'mod'.

Let `z` be a Variable List and `x` be a Variable.

 ©1996 *B-Core (UK) Ltd.*

`z,x`
`((z,x))` is a Variable List.

3.2.18 Generalised Substitutions

Let `x` be a Variable, `z` be a variable List, `P` and `R` be predicates, `E` and `F` be Expressions, and `S`, `T` be Generalised Substitutions.

`[S]P`
A predicate obtained by replacing the variables in `P` according to the rules below.

`[x:= E]F`
An expression obtained by replacing all free occurrences of `x` in `F` by `E`.

`z\A`
Non-freeness: `z` is not free in `E`, i.e. there are no free occurrences of `z` in the Predicate or Expression `A`.

`x:= E`
Simple substitution. Substitute `E` for `x` in a Predicate or Expression formula. (Note that the applicability of a simple substitution on a formula is limited by non-freeness conditions when the formula is a quantified expression or a set comprehension).

`x,y:= E,F`
Simultaneous substitution. Substitute several Expressions for several Variables.

`x:=E || y:=F`
Simultaneous substitution. A form equivalent to the above simultaneous substitution.

`P | S`
Pre-conditioning of `S` by `P`. `[P | S]R = P & [S]R`.

`P ==> S`
Guarding of `S` by `P`. `[P ==> S]R = P => [S]R`.

`S [] T`
Choice between `S` and `T`. `[S [] T]R = [S]R & [T]R`.

`@z.S`
Unbounded choice. `[@z.S]R = !z.[S]R`

`S;T`
Sequencing. `[S;T]R = [S][T]R`.

`skip`
No-op.

 ©1996 *B-Core (UK) Ltd.*

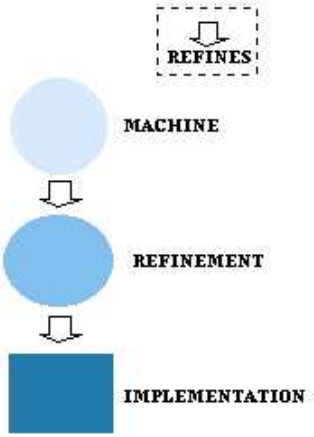


Figure 3: Refinement Structure

3.3 Structuring Developments

3.3.1 Refinement

The fundamental relationship between entities in the B-Method is refinement.

See figure 3.

The REFINEMENT clause is used in REFINEMENTs and IMPLEMENTATIONs to establish the refinement dependency.

An implementation is a particular form of refinement in which an implementable subset of AMN is used.

Proof obligations are associated with refinement to establish that the refining entity meets the constraints of the refined entity.

It is a matter of principle that REFINEMENTs and IMPLEMENTATIONs are only visible through the MACHINE that specifies them. MACHINEs are external interfaces, and it is only MACHINEs that can be cited in the structuring clauses: INCLUDES, USES, IMPORTS and SEES.

It is via this principle that the right separation of concerns is established for separate implementability and other information-hiding issues.

 ©1996 *B-Core (UK) Ltd.*

3.3.2 Structuring Machines

3.3.3 Purpose

The goal of structuring a machine is:-

- to aid in understanding of the specification;
- to factorise commonly used structures;
- to promote reuse;
- to decompose the problem of proving consistency.

It is not the primary purpose of structuring a machine to achieve separate development of components. This is best done by structuring implementations (see below). It is rather to achieve separation of proof concerns.

3.3.4 Including Machines

The INCLUDES clause is the main mechanism for structuring large machines. It allows the abstract system state to be divided into several independent parts, each encapsulated by a separate included machine. These parts can then be combined into a single 'including' MACHINE using the INCLUDES clause.

See figure 4.

INCLUDES is transitive, i.e. included machines may include other, and so on, as illustrated in the diagram. This allows specifications to be constructed hierarchically. No cycles in the inclusion relationship are allowed.

If an abstract state can be partitioned naturally into several independent parts, then it is an advantage to use this structuring technique. A measure of this is whether the operations on the abstract state can be built from operations defined for the individual parts.

After identifying the independent parts, an included machine is written for each part. Each included machine will introduce its own state variables. To avoid name clashes in the including machine, it is best to make the name spaces of the included machines disjoint.

Several 'copies' of the same machine can be included by using machine renaming, which has the effect of making the name spaces of the two copies disjoint. Note, however, that names of SETS and CONSTANTS do not participate in the renaming.

A machine is only allowed to change its local state. So if an operation of the including machine requires the state for an included machine to change in a certain way, it can only do this by invoking an operation on the included machine provided for this purpose.

 ©1996 *B-Core (UK) Ltd.*

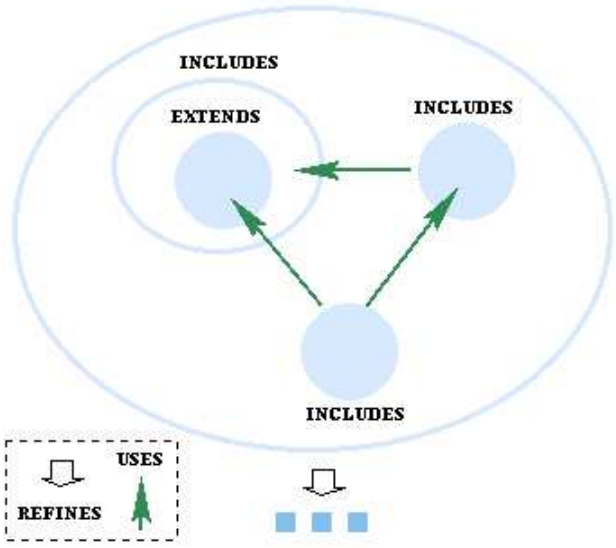


Figure 4: Includes Structure

The including machine state is the combined state of all its included machines. The operations which change the state are built by combining operations of the included machines.

Query operations are those that do not change the state of a machine. In an including machine, these can be written by referring directly to any of the included variables. There is therefore no point in writing query operations in machines that are only going to be included in another.

The including machine can add to the combined state by defining its own state variables, whose names must be distinct from those it is including.

While all the operations of the included machines are available to the including machine, none of these operations are part of the interface of the including machine — unless this is explicitly specified using the PROMOTES clause.

The PROMOTES clause allows us to specify that an included operation is part of the interface of the including machine. It is simply a list of operations which we want to promote. The EXTENDS clause is used instead of INCLUDES if we want to PROMOTE all included operations.

INCLUDES and EXTENDS can only be used in MACHINEs. The PROMOTES clause can be used in an IMPLEMENTATION, but it promotes imported operations rather than included operations.

 ©1996 *B-Core (UK) Ltd.*

Included machines cannot be refined. The outermost including machine is the focus of refinement and implementation.

3.3.5 Using Machines

The INCLUDES clause allows one machine to have exclusive knowledge of another. During specification construction using INCLUDES (or EXTENDS), it is often convenient for included machines to have shared knowledge of the variables of another included machine. This can be achieved by using USES. The using machine can reference all the variables of the used machine, but cannot modify the state of that machine.

The variables of the 'used' machine do not become part of the state of the using machine. They can, however, be quoted anywhere inside the 'using' machine except on the left-hand side of a substitution.

You can name a parameterised machine in a USES clause, but you are not allowed to provide parameters. Nor can you reference the formal parameters of the used machine in the 'using' machine.

Used machines and their 'using' machines must both be included in the same 'including' machine in the overall specification structure. At this level, all formal parameters of included machines are provided.

USES is not transitive, i.e. knowledge of variables from a used machine does *not* transfer from the using machine to any other using, including, seeing or importing machine.

3.3.6 Non-Determinism in Included Machines

Unbounded non-determinism in the operations of included machines should be avoided. Parameterisation should be used instead.

The reason for this is that it is not possible to have any influence on the behaviour of non-deterministic operations from outside the machine. Through parameterisation, control can be exerted from without.

3.3.7 Specification Structure and Proof

By structuring a specification using INCLUDES, not only are proofs of consistency correspondingly smaller, but the restricted impact of changes to the specification means that fewer re-proofs have to be performed when local changes are made.

In the including machine, the proof obligations associated with included machines are assumed to be valid. Only if a new invariant is given in the including machine is there extra work to be done: the operations of the including machine have to preserve the new invariant.

 ©1996 *B-Core (UK) Ltd.*

3.3.8 Viewing the Structure of Specifications

The B-Toolkit Overview Tool can be used to view the structure of a specification, by selecting the Specification Overview option from the Overview menu (see page 16).

3.3.9 Structuring Designs

3.3.10 Purpose

The goal of structuring a design is:-

- to provide a layered approach to design;
- to achieve separate implementability;
- to allow code sharing;
- to ease the difficulty of proof of refinement;
- to encourage reuse of system library and team library machines.

3.3.11 Importing Machines

The IMPORTS clause is the main mechanism for structuring designs. An IMPLEMENTATION imports specifications of lower layers of design, allowing the design to be hierarchically structured.

See figure 5.

The full information hiding enforced by importing allows the imported machines to be separately developed.

When a machine is imported, it is also instantiated by providing values for its formal parameters.

The 'leaves' of the development tree in the B-Method are always pre-implemented library machines. Where more than one instance of a library machine is imported, the library machine machines have to be renamed so that the name is unique across the whole development. Thus no sharing of code can be achieved through importing alone.

3.3.12 Seeing Machines

The SEES clause is used in REFINEMENTs and IMPLEMENTATIONs for the purpose of sharing code, and giving access to shared data.

 ©1996 *B-Core (UK) Ltd.*

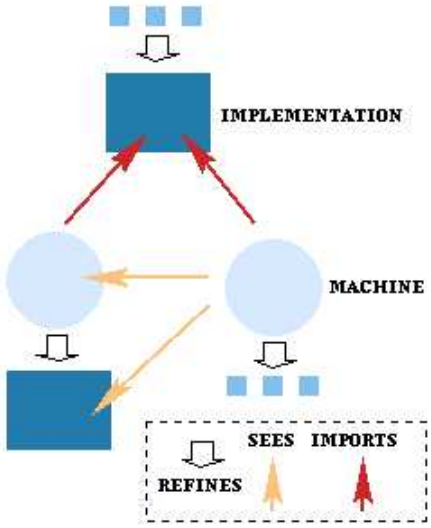


Figure 5: Imports Structure

Machines that are imported once somewhere in a development can be 'seen' elsewhere in the development by citing the 'seen' machine in a SEES clause. The code for the seen machine is linked only once in the development.

This establishes a 'one writer, many readers' sharing scheme. Where the machine is imported, all operations can be invoked in the importing machine to effect state changes in the seen machine, corresponding to 'write' access. Where the machine is seen, only query operations can be invoked, corresponding to 'read-only' access.

A stateless machine is one which has no abstract state variables. Since they are stateless, all their operations are queries, and citing them in a SEES clause gives access to them all.

Stateless machines are typically used for sharing separately implemented system-wide types. Examples of these can be found in the System Library, such as SCALAR, BOOL, STRING. These machines contain mathematics and operations, and have an implementation.

Stateless machine may also be operation-less. These can be used for sharing a particular mathematical context between several machines. Rather than copying the same definitions into the CONSTANTS and PROPERTIES clauses of several machines, the definitions can be placed in a separate state-less (and operation-less) machine. This machine can then be cited in the SEES clause of every machine that needs the definitions.

The SEES clause can be used to share such machines at any layer of development. In fact, since SEES is not transitive, the sees machine will often have to be cited in the SEES

 ©1996 *B-Core (UK) Ltd.*

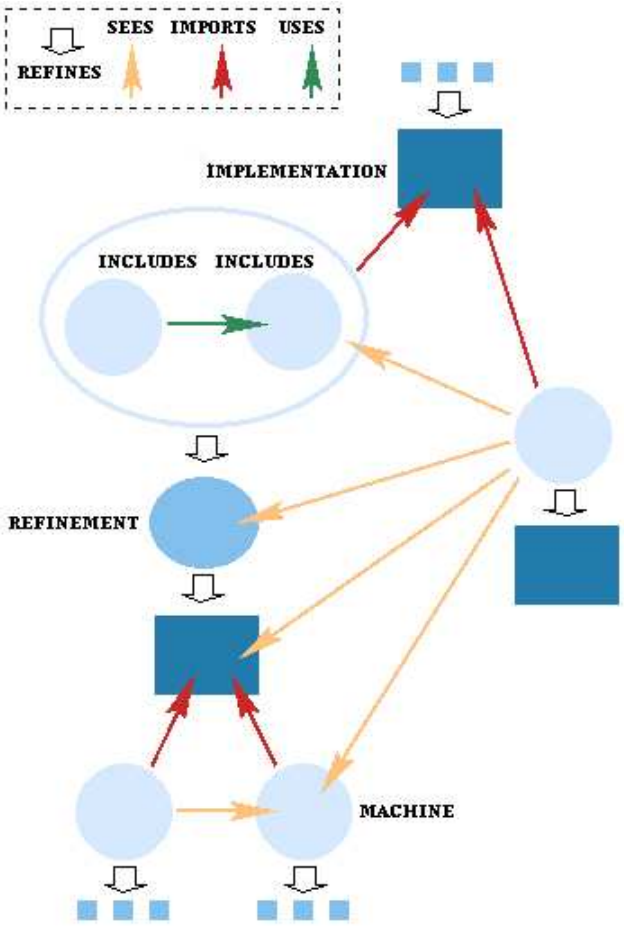


Figure 6: Sees Structure

clause of several machines down through the hierarchy.

See figure 6.

By importing a stateless machine once and seeing it many times, only one copy of the code is present in the final system. If a machine is imported twice in a development, name clashes will arise during linking.

Operation-less machines may not need refining. Since it is just mathematics that they contain, the definitions may simply be 'programmed away' in the algorithms used in the main development. However, from a book-keeping point of view, a dummy IMPLEMENTATION has to be present before linking can take place. This dummy implementation can take the following form:-

```
IMPLEMENTATION
```

 ©1996 *B-Core (UK) Ltd.*

```
    My_Maths_I
REFINES
    My_Maths
END
```

If such an implementation has not been provided, the user will be notified if he attempts translation.

As with the USES clause above, you can name a parameterised machine in a SEES clause, but you are not allowed to providing parameters. Formal parameters are provided where the 'seen' machines is imported in the development.

SEES is not transitive, i.e. knowledge of variables from a seen machine does *not* transfer from the seeing machine to any other using, including, seeing or importing machine.

3.3.13 Hierarchical Design

The IMPORTS clause allows a development to be structured into layers of specification/implementation pairs. The IMPLEMENTATIONs of one layer import the specifications of the layer below.

See figure 7.

It is this capability that makes medium to large-scale software development possible with the B-Method and B-Toolkit. Monolithic refinement of a single large specification into code is impractical. Hierarchical design allows complex systems to be specified abstractly, and implemented in detail, all within the same formal framework.

The specifications of each descending layer introduce more and more detail. The structure of the top level machine does not have to be strongly reflected in the structures of the machines in the next level down. Thus in the top level the most abstract machines can be presented in the most convenient way for specification purposes, whilst in the lower layers, they can be organised in the best way for implementation concerns, such as space and time efficiency.

There is no prescribed way of proceeding with a hierarchical design. Top-down or bottom-up designs are possible, and a mixture of the two is usually adopted.

The following is a typical approach:-

1. We begin by specifying the machines of a middle layer, in which the essential functionality of the system is captured. Animation and proof of consistency are used to establish the desired properties of the system at this stage.

2. A lower layer of machines are specified, in which the previous layer of step 1 can be implemented. These machines may be drawn from the System Library or Team

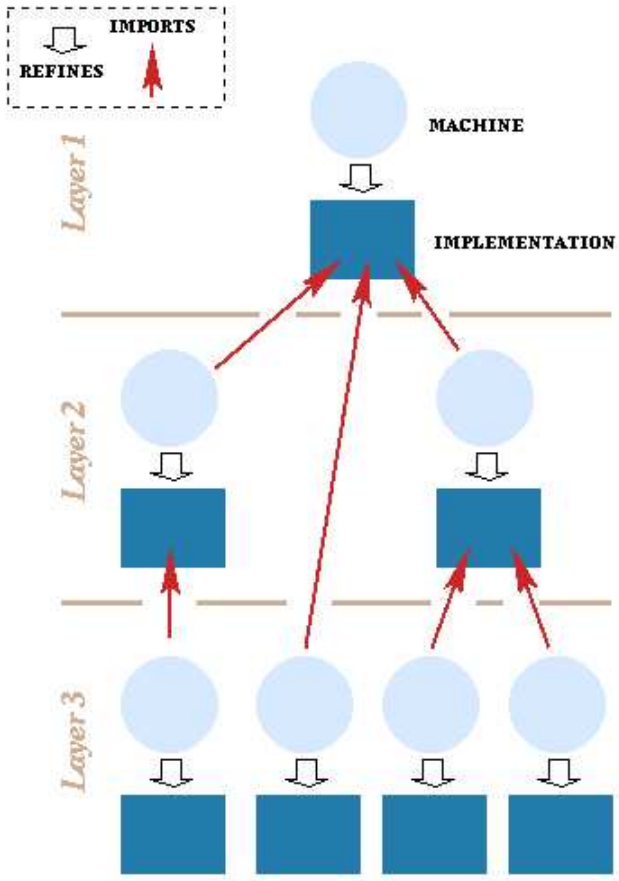


Figure 7: A Typical Hierarchical Design

©1996 *B-Core (UK) Ltd.*

Library, or invented for a specific purpose. Again, Animation and proof of consistency are used here.

3. The layer specified at step 1 is now implemented in terms of the lower layer at step 2. Proofs of refinement are carried out to verify the design so far.

4. Development can proceed downwards towards code in a similar fashion, repeating steps 2 and 3.

5. The final step is to construct a top layer which presents the specific application's functionality through an appropriate interface. Where the machines specified in the middle layer of step 1 may, for convenience of specification, have preconditions, the top layer is usually robust. Code is added to check that preconditions are preserved before invoking operations from the middle layer.

The middle layer can be used to provide an abstract instruction set specially designed to support a range of applications in the problem domain. The top layer then uses the generic functionality to implement a particular application. This approach encourages the maximum of reuse of specifications, designs and code.

A top layer can simply be a 'main' program (an operation called `main`), which must exist before code can be executed.

One way of constructing such a top layer is by using the Interface Generator (see page 11), which constructs a default main program with an interface that allows the operations of the middle layers to be invoked.

3.3.14 Design Structure and Proof

By structuring a development using IMPORTS, it is possible to 'divide and conquer' the problem of proof of refinement, as well as that of development.

The proofs of consistency of imported machines can be assumed in proofs concerning the importing machine.

Not only are proofs of refinement correspondingly smaller, but the restricted impact of changes to the development means that fewer re-proofs have to be performed.

3.3.15 Viewing the Structure of Designs

The B-Toolkit Overview Tool can be used to view the structure of a design, by selecting the Design Overview option from the Overview menu.

In this option, the Overview Tool displays the hierarchical structure established by the IMPORTS clauses in the development.

 ©1996 *B-Core (UK) Ltd.*

3.4 ASCII Symbols for the Abstract Machine Notation

3.4.1 Table of relation types

RELATIONS Symbol	Mathematical type	Cardinality Domain/Range	Optionality Domain/Range
<->	relation	many/many	optional/optional
+->	partial function	many/one	optional/optional
-->	total function	many/one	mandatory/optional
+->>	partial surjection	many/one	optional/mandatory
-->>	total surjection	many/one	mandatory/mandatory
>+>	partial injection	one/one	optional/optional
>->	total injection	one/one	mandatory/optional
>->>	bijection	one/one	mandatory/mandatory

3.4.2 AMN Symbols

&	logical and, conjunction
or	logical disjunction
=>	logical implication, implies
<=>	logical equivalence, if and only if
not	logical negation
!	universal quantification, for all
#	existential quantification, there exists
=	equality
/=	inequality

 ©1996 *B-Core (UK) Ltd.*

```
==                  rewrite rule

|->                 ordered pair, maplet

:                   set membership, belongs to, element of

/:                  set non-membership

<:                  set inclusion, subset of

/<:                 set non-inclusion

*                   Cartesian product

\/                  set union

/\                  set intersection

-                   set difference

{}                  empty set, null set

POW                 powerset

POW1                set of all non-empty subsets

FIN                 set of all finite subsets

FIN1                set of all non-empty finite subsets

{E}                 singleton set

{E,F}               enumerated set

union               generalised union

inter               generalised intersection

UNION(z).(P | E)    generalised union

INTER(z).(P | E)    generalised intersection
```

 ©1996 *B-Core (UK) Ltd.*

>	strict inequality, greater than
<	strict inequality, less than
>=	inequality, greater than or equal
<=	inequality, less than or equal
NAT	set of natural numbers
NAT1	set of non-zero natural numbers
max	maximum
min	minimum
+	addition, plus
-	difference, minus
*	product
/	division
mod	remainder of integer division
..	interval
card	set cardinality
SIGMA	summation
PI	product
dom	domain of relation
ran	co-domain of relation
;	relational composition
circ	relational composition
id	identity relation

 ©1996 *B-Core (UK) Ltd.*

`<|`	domain-restriction
`|>`	range-restriction
`<<|`	anti-domain-restriction
`|>>`	anti-range-restriction
`r~`	inverse
`r[s]`	image
`<+  +>`	overriding
`><`	direct product
`||`	parallel product
`iterate(r,n)`	iteration
`closure(r)`	reflexive transitive closure
`prj1`	projection
`prj2`	projection
`%(z).(P | E)`	lambda abstraction
`f(x)`	function application
`<>`	empty sequence
`seq(S)`	set of finite sequences over S
`iseq(S)`	set of injective sequences over S
`seq1(S)`	set of non-empty sequences over S
`perm(S)`	set of bijective sequences over S
`^`	sequence concatenation

 ©1996 *B-Core (UK) Ltd.*

```
->              prepend

<-              append

[E]             singleton sequence

[E,F]           sequence enumeration

size            sequence size

rev             sequence reverse

/|\             sequence projection (prefix)

\|/             sequence projection (suffix)

first           first sequence element

last            last sequence element

tail            sequence tail

front           sequence front

conc            generalised concatenation

x:=E            simple substitution

x,y:=E,F        simultaneous substitution

x::E            membership postconditioned substitution

x: P            general postconditioned substitution

S ; T           substitution S followed by substitution T

skip            no-substitution

oo <-- op(ii)   parameterised substitution with input ii, output oo
```

 ©1996 *B-Core (UK) Ltd.*

3.4.3 Generalised Substitution Language

`[S]P`	substitution application to a predicate P
`[x:= E]F`	simple substitution on expression F
`z\A`	non-freeness of z in A
`P | S`	pre-conditioning of S by P
`P ==> S`	guarding of S by P
`S [] T`	choice between S and T
`@z.S`	unbounded choice

 ©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

B-TOOLKIT USERS' MANUAL

SECTION 4

Construct Management

©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

4 Construct Management

4.1 The Introduce Tool

This option is invoked from the Tool Bar and provides the usual method of introducing a new construct into a development.

The name of each construct should not already be in use (irrespective of the type of the construct); the only exception to this is for an interface, which shares the name of the specification it interfaces.

There is a restriction on the length of the name of a construct: the maximum length is 20 characters (12 for base constructs) excluding the extension.

4.1.1 Introduce New Construct

This facility enables the introduction of a new construct in template form (e.g. with appropriate keywords included in the construct) into the development; the construct name is prompted for, and the construct created in the user's SRC directory and opened for editing:

This option provides a sub-menu of twelve options; the following seven:

- Machine
- Refinement
- Implementation
- Base
- Enumeration
- Interface
- Document

each causes a file containing appropriate keywords to be created in SRC.

The remaining five:

- Refinement of analysed construct
- Implementation of analysed construct

 ©1996 *B-Core (UK) Ltd.*

- Enumeration of analysed machine(s)
- Interface of implemented machine
- Document of configured constructs

cause the toolkit to prompt for the appropriate construct(s) required, after which a file containing as much information as possible (e.g. all operation names with skeleton operations in the first two cases) be created in SRC.

4.1.2 Introduce construct(s) from SRC

This option produces a menu of constructs currently in the user's SRC directory (constructs whose name is already in use will not be offered).

If the construct parses, it will be committed to the CFG directory; if it does not parse, an appropriate parsing error message is given in the Display Area and the construct is opened for editing together with an invitation to introduce after editing.

After successful introduction, the construct will appear in red the Constructs Area of the appropriate Environment Window(s).

4.1.3 Introduce construct from SLIB

If a construct is introduced from the system library or the team library, the library specification will be copied into the development CFG directory and the construct will appear in blue in the Constructs Area of the appropriate Environment Window(s).

In the case of a renamable SLIB machine, the user will first be prompted for the rename prefix (the length of the renamed construct should not exceed 20 characters).

Since library machines, like generated machines, are guaranteed to be syntactically correct and well-typed, related analysed forms and type information files are also copied to the ANL and TYP directories. The status of these constructs is reflected in the various B-Toolkit environments by the command buttons that are active.

Further, when a library construct is introduced, its matching code (renamed, if necessary) is also compiled and copied to the CDE/C directory.

The mathematical content of these library machines may NOT be changed and committed to the system (i.e. they are 'read-only' in that sense). It is, however, possible to commit annotation changes.

4.1.4 Introduce construct from TLIB

This works as for introduction from SLIB. Only machines that have been previously ex-

 ©1996 *B-Core (UK) Ltd.*

ported to the Team Library can be introduced from the Team Library.

The environment variable BTEAMLIB must be set for this function to work.

©1996 *B-Core (UK) Ltd.*

4.2 The Editor

The 'opn- quit-editor' option may be invoked from all environments selectable from the Top Bar, and for all constructs. It represents a toggle for opening/closing (quitting from) a construct edit; if the construct is currently open, the option label is indented to the right.

4.2.1 Open Editor

A copy of the CFG file is first made in the SRC directory; if a file of that name currently exists in SRC, and differs from that in CFG, the following options are offered:

- `Open SRC file`
  opens the SRC file, when edits may subsequently be committed in the usual way.
- `Open CFG file`
  opens the CFG file and in so-doing overwrites the edited SRC file (which is thus lost); edits may subsequently be committed in the usual way.
- `Show difference`
  displays the UNIX diff between the two files.

If the environment variable BEDIT is set, the edit is invoked through the command '$BEDIT file', otherwise through 'xedit file'.

4.2.2 Quit Editor

If the currently-edited SRC file is the same as the CFG file, the following prompt is displayed:

```
Quit editor requested, but no changes saved!

'Cancel' if you want to save any changes ...
```

to ensure that any edits made which have not been saved are not unintentionally lost. The 'OK' option causes the edit to be killed, with no commit therefore being necessary.

However, if there is a difference between the SRC and CFG files (i.e. the edited file differs from that which is currently committed), the following choice is offered:

- `Commit edits`
  updates the CFG directory, and in so doing causes any processing of the construct

 ©1996 *B-Core (UK) Ltd.*

to be lost; further, processing of any dependent constructs is also lost (but you will be warned in this case before the commit is actually made). If the difference is annotational, the processing lost is restricted to marked-up documents.

- `Discard edits`
  removes the edited SRC file from the file system.

- `Save in SRC`
  does not commit the edited file, but leaves it in the SRC directory for later use.

- `Show difference`
  displays the UNIX diff between the SRC and CFG files.

It is also possible to commit the changes of an edited construct through the Close, CommitEdits, Commit, Remake, Analyser, POGenerator and Mini-Remake tools.

©1996 *B-Core (UK) Ltd.*

4.3 Close

This option is invoked from the Tool Bar, and presents a selection of currently open constructs (an error results if there are none). Unlike similar menus (with the exception of those for Commit and Remake) all constructs are 'pre-selected' so that pressing the 'OK' button has the effect of selecting all. The option then proceeds as though the 'opn -quit-editor' button had been pressed for each selected construct.

It is also possible to commit the changes of an edited construct through the Editor, CommitEdits, Commit, Remake, Analyser, POGenerator and Mini-Remake tools; see Top Bar Help for the appropriate tool.

©1996 *B-Core (UK) Ltd.*

4.4 Commit

This option is invoked from the Tool Bar, and presents a selection of currently open constructs whose edited SRC files differ from their currently-committed CFG files (an error results if there are none). Unlike similar menus (with the exception of those for Close and Remake) all constructs are 'pre-selected' so that pressing the 'OK' button has the effect of selecting all.

The option then proceeds as though the 'cmt' button had been pressed for each selected construct - see Top Bar Help for CommitEdits.

It is also possible to commit the changes of an edited construct through the Editor, CommitEdits, Close, Remake, Analyser, POGenerator and Mini-Remake tools.

©1996 *B-Core (UK) Ltd.*

4.5 ResetConstruct (rst)

This option is invoked from the Main Environment and is applicable to all non-library non-generated constructs on which some processing has been done.

It provides an alternative to the other global Reset options, limiting the effect of the reset to a single construct and its dependents (if any - in this case the dependent list will be displayed in the Display Area, and you will be prompted to continue).

©1996 *B-Core (UK) Ltd.*

4.6 Rename

The Rename facility is available under the Tool Bar and provides a limited capability for the renaming of a configured construct.

It is not possible to rename the following constructs:

- a library construct
- a construct which has been generated
- a construct which is currently open for editing
- a machine/refinement/implementation which is currently analysed
- a base/enumeration/interface which is currently generated
- a document which is currently marked up

It is not possible to rename a construct to the same name as that of a currently-committed construct (irrespective of the type of the construct).

In a successful rename, the construct name and filename are changed.

©1996 *B-Core (UK) Ltd.*

4.7 Remove

The Remove facility is invoked under the Tool Bar, presenting a selection of removable constructs from which the user may select one or more. If a construct selected for removal has dependencies, these are displayed in the Display Area and a prompt to continue is displayed; removing such a construct results in dependent constructs being reset.

The facility allows the user to 'de-introduce' a construct i.e. to remove it from configuration. A removed construct is moved from the CFG to the SRC directory. Any removed construct may be re-introduced to the development through the 'Introduce construct(s) from SRC' option - see Introduce.

It is not possible to remove generated constructs - the generating construct itself must be removed, when the generated constructs are completely removed from the file system - since they may be re-generated.

Removal of a library construct will also result in it being completely removed from the file system (since it may be re-introduced from one the library introduce options - see Introduce).

©1996 *B-Core (UK) Ltd.*

4.8 Status (sts)

This option is invoked from the Main Environment and is applicable to all machines, refinements and implementations.

It allows the user to determine the current state of a construct under configuration; status information comprises:

- whether or not the construct is currently analysed; if so, the following information is provided:
  - the number of proven/unproven proof obligations associated with the construct at each proof level
  - whether or not the construct is currently marked-up
  - in the case of an implementation, whether or not it is currently translated
  - a list of dependent constructs
  - a list of subordinate constructs
  - the abstraction/refinement/implementation list

(A construct `A` is 'dependent' on a construct `B` if `A` SEES, USES, INCLUDES, IMPORTS or REFINES `B`; in those cases `B` is said to be a 'subordinate' construct to `A`).

 ©1996 *B-Core (UK) Ltd.*

4.9 Hypertext View of a Development

The Hypertext Tool is invoked from the Tool Bar in any environment, and produces a selection of currently-analysed constructs together with a summary page of those constructs. Each represents a hypertext version of the corresponding file in the CFG directory, and is automatically created (in the HTX directory) for each of the following:

- on analysis of a user-introduced machine, refinement or implementation
- on analysis of a base-generated machine or implementation
- on analysis of an interface-generated machine or implementation
- on introduction of an SLIB construct
- on introduction of a TLIB construct
- on enumeration

4.9.1 Referencing constructs

In each referencing construct, a hypertext link is created to the relevant subsidiary construct for the appearance of each of the following:

- a machine, refinement or implementation name
- a set name
- a constant name
- a variable name
- an operation name

4.9.2 Referenced constructs

Every construct `construct_name.ext` in the HTX directory has the filename `construct_name.ext.html`; destination hypertext links are constructed using the following convention:

- for the construct itself:
  - `<A NAME="MACHINE">MACHINE</A>`
  - `<A NAME="REFINEMENT">REFINEMENT</A>`

 ©1996 *B-Core (UK) Ltd.*

  - `<A NAME="IMPLEMENTATION">IMPLEMENTATION</A>`

- for its sets:
  - `<A NAME="SETS">SETS</A>`
- for its constants:
  - `<A NAME="CONSTANTS">CONSTANTS</A>`
- for its variables:
  - `<A NAME="INVARIANT">INVARIANT</A>`
- for each of its promoted operations:
  - `<A NAME="PROMOTES">PROMOTES</A>`
- for each of its own operations:
  - `<A NAME="op_name">op_name</A>`

Thus it is possible to link other documents (for example informal requirements) to these HTX documents by reference to the above convention. For example the variable `var1` of `fifi.mch` would be referenced:

```
<A HREF="fifi.mch.html#INVARIANT">var1</A>
```

assuming that the referencing construct also resides in the HTX directory.

 ©1996 *B-Core (UK) Ltd.*

4.10 Development Overview

The Overview Tool is invoked from the Tool Bar in any environment, and is applicable to all specifications, refinements and implementations that are currently analysed.

It provides hypertext-like facilities in picture form, enabling the user to navigate through a development through a push-button interface.

The following options are initially presented:

- Design overview
- Specification overview
- Construct

The last option provides a sub-menu of all constructs that may currently be browsed (i.e. all specifications, refinements and implementations that are currently analysed), with subsequent selection resulting in a detailed overview of the selected construct.

The specification and design overviews are presented in a layered horizontal manner; selecting any construct will result in a detailed overview of that construct, when the diagram is displayed in vertical form, with seen/ used constructs (if any) to the left of the overviewed construct and included constructs (when overviewing a specification) or imported constructs (when overviewing a design) on the right:

See figure
8.

Selecting any of the constructs will result in an overview of that construct.

See figure
9.

Subsequently electing the main (highlighted) construct will then popup a selection comprising the abstractions /refinements of that construct (if any); subsequent selection will result in an overview of that construct; the selection also provides the facility to open the main construct for editing (if it is not already open):

See figure
10.

Each overview provides the following options:

- `Previous`
  restores the previous overview (where applicable).

 ©1996 *B-Core (UK) Ltd.*

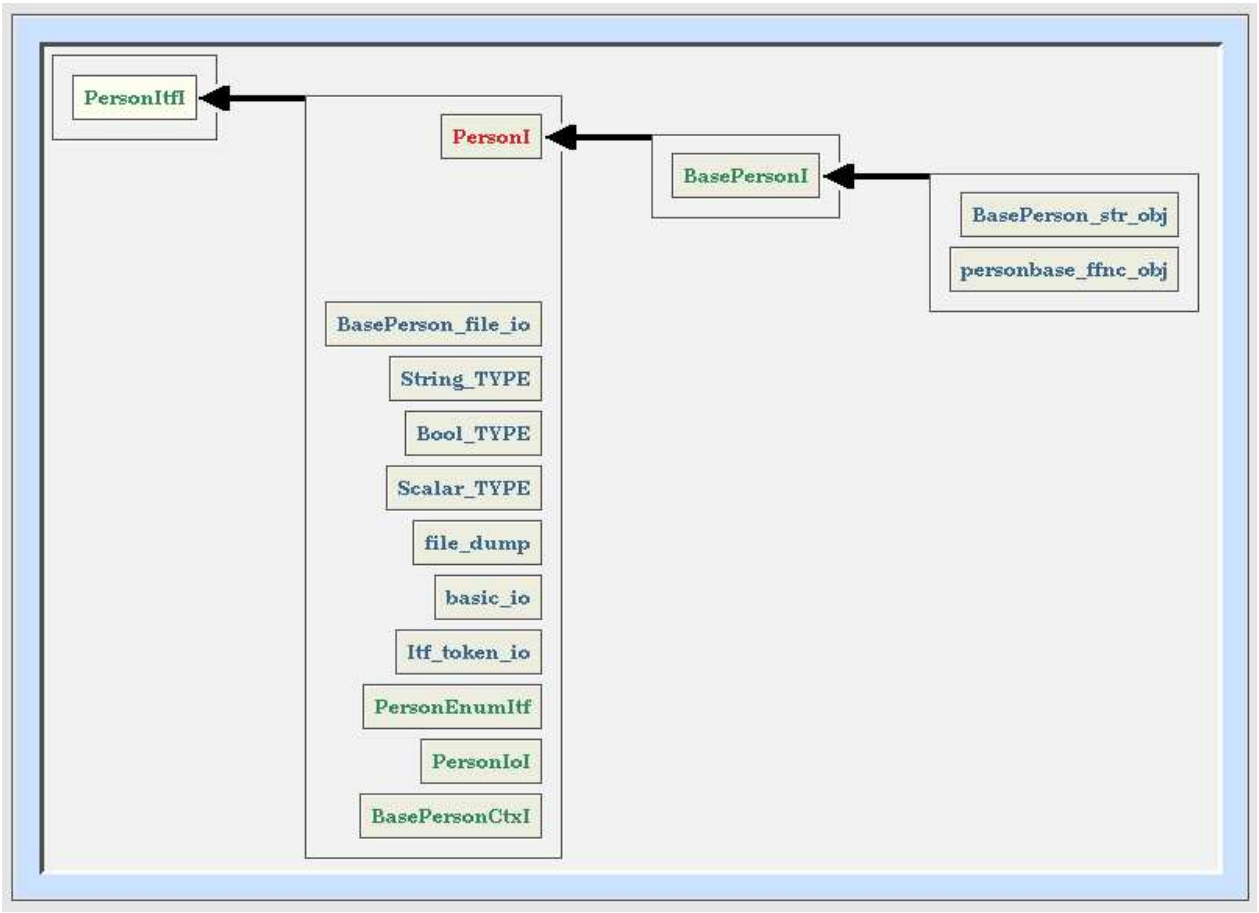


Figure 8: Browsing a Design

©1996 *B-Core (UK) Ltd.*

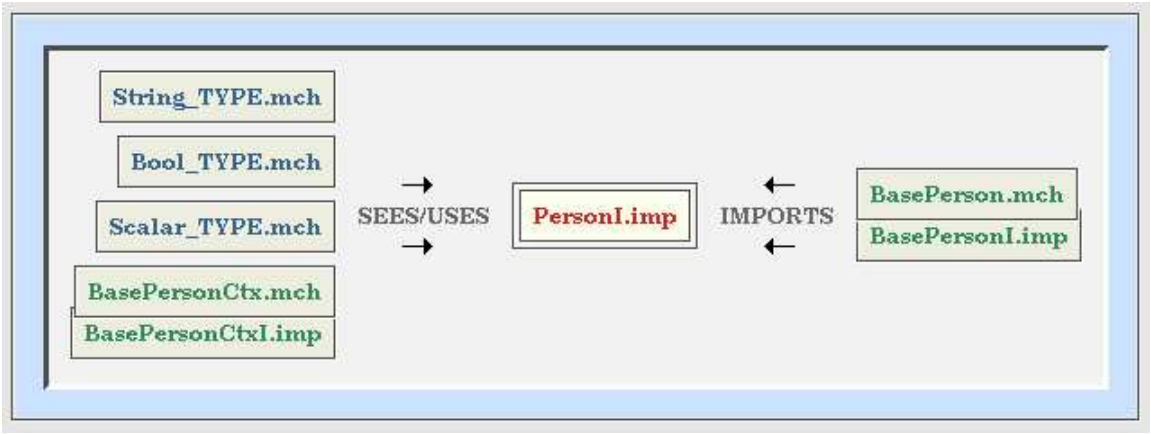


Figure 9: Browsing a Construct

Figure 10: Browsing an Abstraction/Refinement

 ©1996 *B-Core (UK) Ltd.*

- `Specification`
  provides an overview of the specification construction.
- `Design`
  provides an overview of the design.

 ©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

B-TOOLKIT USERS' MANUAL

SECTION 5

Analysis

©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

5 Analysis

5.1 The Analyser (anl)

Analysis is the first stage of processing necessary for a MACHINE, REFINEMENT or IMPLEMENTATION and is a prerequisite for the application of the POGenerator (pog) — and hence the Provers — the Animator (anm) and Documentation Tool (dmu).

The POGenerator automatically invokes the Analyser for a construct which has not been analysed. Thus for a newly-introduced MACHINE, REFINEMENT or IMPLEMENTATION only the Editor, Analyser and POGenerator options are available.

The Analyser (anl) is invoked from the Main Environment, and is applicable to all specifications, refinements and implementations not currently analysed.

Analysis comprises three distinct phases: normalisation, syntax-checking and type-checking. The Normaliser produces an internal form — the analysed form — for the construct, expanding subordinate constructs where necessary. The SyntaxChecker performs syntactic checks on the construct, and the TypeChecker performs checks for type correctness.

If the Analyser succeeds, an internal form of the construct is created and stored in the directory ANL, and a summary of the type information (in internal form) is placed in a "Type File" in the TYP directory, for use by the other tools.

The dependencies between parent and subordinate constructs are built up and recorded by the Analyser. Any record of such a dependency will be removed when the parent or one of its subordinates has been edited such that the mathematical content is changed, and the change committed (by one of the Commit, Close or Remake tools); the parent's analysed form (and all subsequent processing on the construct) will also be lost (although it may be automatically reconstructed using the Remake tool).

If an error is detected during analysis then the MiniRemake option is offered.

5.1.1 Edited Files

If the Analyser is requested for a construct which is being currently edited, and the edited version (i.e. that version in the SRC directory) differs from that of the committed version (in the system CFG directory), three options are offered:-

- Proceed after committing
- Proceed without committing
- Show Difference (between the SRC and CFG files)

 ©1996 *B-Core (UK) Ltd.*

Accepting the commit option will have the same effect as that described for the Commit tool.

5.1.2 Subordinate Constructs

The Normaliser will need to access the analysed form for all constructs subordinate to that which is being currently analysed. (A construct 'A' is dependent on a construct 'B' if 'A' SEES, USES, INCLUDES, IMPORTS or REFINES 'B.' Conversely, 'B' is said to be a subordinate construct of 'A'.)

If a subordinate construct is configured but not currently analysed, the subordinate construct will first be analysed; this process is recursive.

If a subordinate construct is not configured, the Normaliser will search first in the SLIB directory (taking account of any 'dot rename' prefixes), then the TLIB directory, and finally the SRC directory; if found, it will first be introduced and then analysed.

If the above search is not successful, the Normaliser will check to see whether the subconstruct is generated from an ENUMERATION; if the appropriate enumeration is configured, it will be generated. If the enumeration is not configured but exists in the SRC directory, it is first committed and the the machine generated.

5.1.3 Type Determination

Each variable (a state variable, a constant, a quantified variable in a predicate or a local variable within an operation) is given a type by the i.e. the tool determines to which set within a pool of sets the variable belongs. The pool of sets consists of the sets given in the SETS clause and in the formal parameter list, together with all other sets which can be constructed from the given sets by means of the power set operation (POW) and the Cartesian product operation (*).

For example:

if `cc` is a constant and if `CSET` is a given set and the PROPERTIES clause contains

1. `cc <: CSET`
   then `cc` is given type `POW(CSET)`
2. `cc = 7`
   then `cc` is given type `NAT`
3. `cc : NAT --> CSET`
   then `cc` is given type `POW(NAT*CSET)`

 ©1996 *B-Core (UK) Ltd.*

if ASET is also given and aa and rel are variables and the INVARIANT clause contains

1. aa <: ASET
   then aa is given type POW(ASET)
2. aa = cc
   then bb is given the same type as cc
3. rel : ASET <-> CSET then rel is given type POW(ASET*CSET)

if bb is a local variable and bb := 1 is the first statement encountered by the TypeChecker then bb is given type NAT.

An error is reported if a type of a variable cannot be determined.

5.1.4 Type Checking

Given the type of all variables then all formulae in which they occur are checked for type consistency (so in the expression aa=bb, aa and bb must be of the same type). An error is reported if a formulae does not type check, and an explanation is given.

Often the error is due to an inconsistent use of externally defined variables or operations. The Find facility will provide the type of those externally defined variables or operations.

 ©1996 *B-Core (UK) Ltd.*

5.2 The Animator (anm)

The Animator is a tool to demonstrate the state transitions of an Abstract Machine specification, achieved through its operations which are invoked interactively through user-provided input. The operations are displayed on screen, together with precondition simplification (where appropriate) with operation output and modified state variables being displayed in a state window.

Animation (anm) is invoked from the Main Environment for all currently analysed specifications, and so all library and generated machines may be animated (a good way to familiarise oneself with their functionality!).

On invoking the tool for the first time, a session window appears, with, for example, the display

```
No file "SRC/fifi.mch.thy" - creating template
```

indicating that no user-supplied theory is provided to supplement the Animator's library of re-write rules, but that a template theory file has been created in the SRC directory (see Providing More Rewrite Rules below); when theory is supplied, the display will read

```
"SRC/fifi.mch.thy" loaded
```

provided the theory files parses, and

```
"SRC/fifi.mch.thy" does not parse - not loaded
```

otherwise, together with information showing the cause of the error.

It is possible to save the state of an animation (see below), and if one or more previously saved state files is found, the following prompt is displayed:

```
Initialise state from file?
```

If you reply 'No', the state of the machine will be initialised according to its specification. You will be asked to instantiate any machine parameters (which must be natural numbers or enumerated sets). The Animator will then calculate the instantiated machine constraints and present them in a simplified form.

A 'Yes' answer provides a menu from which the saved state file may be selected; subsequent selection overrides the initialisation given in the Abstract Machine specification, and instead initialises the state of the Machine to that saved on file. The initial state is then displayed in the state window.

If the specification (or any of its subordinates) contains sets or constants (and the initialise from file option has not been selected), the following prompt is displayed:

 ©1996 *B-Core (UK) Ltd.*

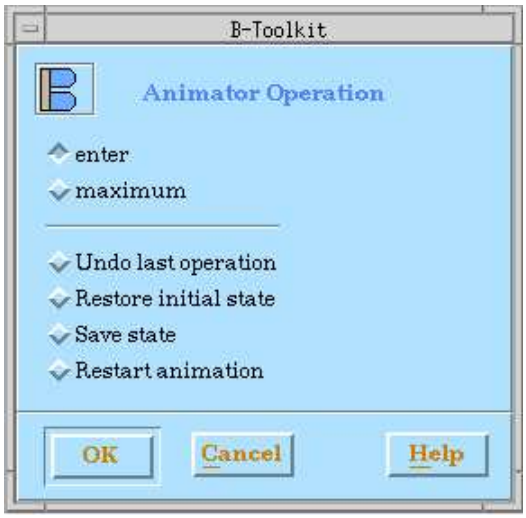


Figure 11: Animator Operations

```
Instantiate deferred sets/constants?
```

If you reply 'No', the sets/constants are left in their abstract, deferred form, whereas a 'Yes' reply will result in a prompt for each set/constant (any of which may still be left in deferred form). After instantiation the machine is initialised according to its specification and the initial state displayed in the state window.

Once animation has started, the operations of the machine are displayed in a menu together with:

- Undo last operation
- Restore initial state
- Save state
- Restart animation

See figure
11.

The 'Undo' option allows you to return to the state prior to the action of the last operation. The 'Restore' operation enables the animation to be restarted with the initial state. The 'Save' option results in a prompt for a filename, with the current state being recorded on that file; at the beginning of a subsequent animations you may then initialise the machine to this state (see above).

 ©1996 *B-Core (UK) Ltd.*

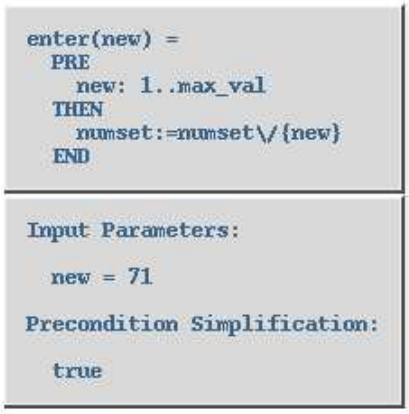


Figure 12: Precondition Simplification

On selection of an operation, its specification is displayed together with its stated precondition. If the operation has input parameters, you will be asked to instantiate them, and the simplified precondition will be displayed:

See figure
12.

You will also be asked to resolve any nondeterminacy (e.g. in an ANY guard). The Animator simplifies, as far as it can, the precondition (or guards) of an operation according to the current state of the Machine. Whatever the value of the precondition or guard, the specified state change is performed, with the value of any output and the final state reported in the state window:

See figure
13.

Errors may be corrected through the 'Undo' operation (see above).

The 'Cancel' option allows you to leave the animation; in the case of non-library and non-generated machines, you are presented with the options:

- Mini-remake
- Edit theory file

The first option allows you to edit your abstract machine specification and return to its animation (with the Analyser being invoked automatically if required).

The second option provides you with an editor in which to provide extra mathematical

 ©1996 *B-Core (UK) Ltd.*

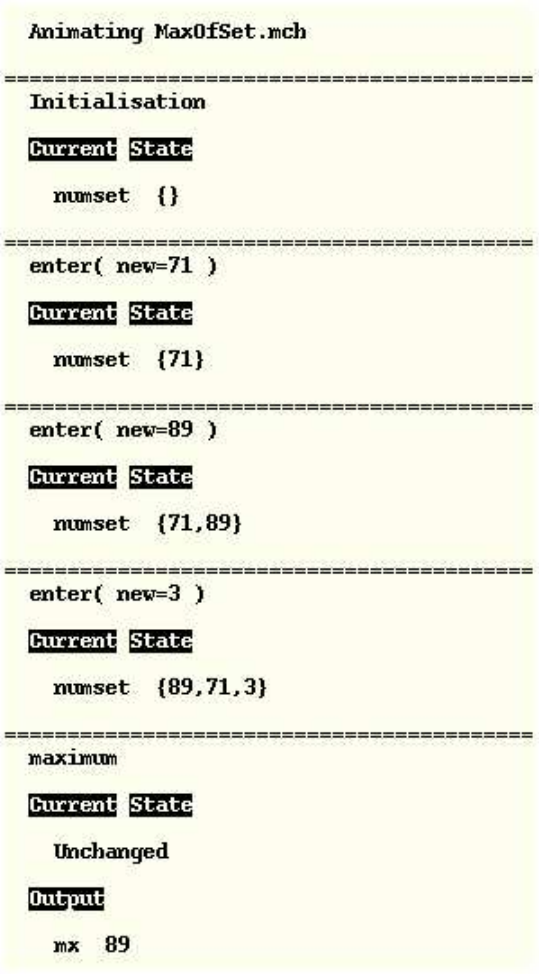


Figure 13: State Display

 ©1996 *B-Core (UK) Ltd.*

simplification rules to assist the animation of the current construct (see below). Animation continues after the file has been edited.

5.2.1 Providing More Rewrite Rules

Simplification of instantiated expressions (outputs, preconditions, guards) is made using a simple set of mathematical re-write rules within the Animator. If the tool is unable to simplify a particular expression, the user can provide appropriate mathematical simplification rules by using the 'Edit theory file' option above. The syntax for the animator theory file is:

```
THEORY UserLibX IS
    rewrite_rule [ ; rewrite_rule ]
END

[
DEFINITIONS
    rewrite_rule [ ; rewrite_rule ]
INCLUSIONS
    incl_file [ , incl_file ]
]
```

The optional DEFINITIONS clause allows macro definitions in the form of rewrite rules to be applied to the rules of UserLibX before use. If the theory file does not exist, it is created by the system with two default macros; a DEFINITIONS clause comprising '?' is equivalent to an empty clause.

The optional INCLUSIONS clause allows other animator theory files to be loaded (and, of course, the process is recursive). Each inclusion should be the name of an animator theory file enclosed in double quotes (for example "SRC/fifi.mch,thy"). An INCLUSIONS clause comprising '?' is equivalent to an empty clause.

The simplification rules of UserLibX should be expressed as B rewrite rules (see section on the B-Platform).

For example:

1. the rule

   ```
    bident(x)
   =>
    [l]^[x] == [l,x]
   ```

   will re-write the expression

 ©1996 *B-Core (UK) Ltd.*

```
[aa,bb,cc]^[dd]
```

to

```
[aa,bb,cc,dd]
```

2. the rule

```
btest(n>1)
=>
fac(n) == fac(n-1)*n
```

will re-write the expression

```
fac(7)
```

to

```
fac(7-1)*7
```


©1996 *B-Core (UK) Ltd.*

5.3 Mini-remake

If an error is detected by the Analyser, POGenerator, Base Generators or Enumerator tools (or if the 'Finish animation' option of the Animator is selected), then the 'Mini-remake' option is presented. If, for example, an error is detected during the processing of 'fifi.mch', an editor is invoked and the following option presented:

- Mini-remake

Selecting 'OK' causes the toolkit to commit your edited changes (save them first!) and to attempt to 'remake' back to the state immediately prior to the error detection, and then to re-process the current job; the cycle is repeated until either no errors remain or the 'Cancel' option is selected

 ©1996 *B-Core (UK) Ltd.*

5.4 Remake

Remake is available on the Tool Bar.

A record of the most advanced state of development reached by each construct is maintained by the system. If the status of any construct (or constructs) is changed (e.g. by editing it) then it is possible to restore the development to its most advanced state from its current state by use of the Remake facility. The Remake tool causes all the necessary tools to be applied to the appropriate constructs automatically until either an error is encountered or the most advanced state is restored.

Two choices are initially offered:

- Remake all constructs
- Remake construct(s)

Selection of the first produces a menu of jobs for all constructs in the development which should be carried out for a full remake; selection of the second produces a menu of constructs from which one or more may be selected, when a menu of jobs relating to those constructs is subsequently produced.

In both cases, the jobs presented depend on the current setting of the Remake Options; each of the following tools may be included in or excluded form the Remake:

- POGenerator
- AutoProver
- ProofPrinter
- DocumentMarkUp
- Translator
- Linker
- ITFGenerator

In both cases, the list of jobs presented is pre-selected, and jobs may be de-selected in the usual way to enable a partial remake to be achieved.

Before the initial two choices are offered, however, if one or more constructs have been edited and the (saved) version in the SRC directory differs from that of the committed version in the CFG directory, the tool will, for each such file, prompt for committing before proceeding in the same manner as the Analyser. Accepting the commit option will have the same effect as that described for the CommitEdit tool.

 ©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

B-TOOLKIT USERS' MANUAL

SECTION 6

Proof

©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

6 Proof

6.1 The B-Platform

The B-Platform is the proof engine that underpins the B-Toolkit, many parts of which are implemented in the B-Theory Language (B-TL) which the B-Platform supports.

B-TL can be viewed as a kind of logic/functional programming language, whose program statements, called "Theories" consist of ordered sets of inference rules. A natural-deduction-style proof engine uses backwards and forwards inference to evaluate programs. Rewriting is treated as a special form of backwards inference.

A tactic language is used to add additional control to the inference mechanism.

B-TL is also the language used to supply additional user theories to the B-Toolkit's Animator and InterProver.

Although no explicit interface to the B-Platform is visible through the B-Toolkit, experienced B-Toolkit users will need to have knowledge of the syntax of B-TL and the proof mechanisms used in the B-Platform.

The B-Platform is best viewed as a "Proof Assistant" in constructing formal proofs. It is not a true "Theorem Prover", because inference rules supplied by the user do not have to be proved in terms of some basic set of axioms. The B-Platform just mechanises the application of rules, and assists with some of the house-keeping.

The B-Platform can be purchased separately from B-Core (UK) Ltd. in their product called the B-Tool (rather than B-Toolkit). It is useful for playing with and learning about proof, as well as for implementing advanced tools such as provers; the B-Platform is also sometimes referred to as the B-Kernel.

For the purposes of B-Toolkit users, a limited knowledge the syntax of a formula in B-TL is all that is required; other aspects of the language are not relevant. What follows below is not intended to be a thorough description of the B-Platform, or a complete syntax for B-TL, but rather a summary of those aspects that are essential to a working knowledge for the B-Toolkit user.

6.1.1 Inference Rules

An Inference Rule in B-TL is a Formula written broadly according to the syntax of the mathematical notation of AMN. An important concept is that of a "Joker", which is used in AMN only in the DEFINITIONS clause.

A Joker can be any single upper or lower-case letter. A Joker is true variable which can be replaced by any Formula. It is to be distinguished from AMN variables, which appear

 ©1996 *B-Core (UK) Ltd.*

as identifiers in B-TL, which have to be at least two characters long.

Identifiers can be modified by appending a `Number`. This is to allow tools to create new variables when copying formulae. In the B-Toolkit, it is used to distinguish between copies of the same state variable in different contexts (inside and outside of a loop, for instance).

Three major types of rules are distinguished, " simple rules", " rewrite rules" and " forward rules", which, despite being syntactically similar, are applied in very different ways. Simple rules and rewrite rules are both used in backwards inference, and so are grouped as "backward rules".

6.1.2 Simple Rules

A Simple Rule has the following syntax:-

```
Simple_Rule    ::= Antecedents => Consequent  |
                   Consequent

Antecedents    ::= Antecedents & Formula      |
                   Formula

Consequent     ::= Formula
```

An example of a simple rule is:-

```
 a <: c &
 b <: c
=>
 a \/ b <: c
```

This rule states, in effect, that a goal of the form

```
a \/ b <: c
```

can be proved if the two goals

```
a <: b &
a <: c
```

can both be proved.

In this example, `a`, `b` and `c` are Jokers which stand for any formula. So applying this rule to the goal

 ©1996 *B-Core (UK) Ltd.*

```
STAFF \/ CUSTOMER <: PERSON
```

has the effect of applying the rule

```
STAFF <: PERSON &
CUSTOMER <: PERSON
=>
STAFF \/ CUSTOMER <: PERSON
```

Simple rules are used for Backwards Inference. They are applied as follows:-

1. The Consequent is matched to the current goal by substituting a formula for each Joker which makes the Consequent identical to the goal. If no such match can be found, the rule cannot be applied.
2. If the match is successful, the same substitution of formulae for Jokers is applied to the Antecedents (if any).
3. The current goal is replaced by a set of new goals which are the new Antecedents (if any).

Note that Simple Rules need not have Antecedents, in which case their application is said to "discharge" a goal: the goal is replaced by an empty set of goals. For example:-

```
a : {a}
```

could match and discharge the goal

```
percy : {percy}
```

6.1.3 Rewrite Rules

A Rewrite Rule has the following syntax:-

```
Rewrite_Rule   ::= Antecedents => Rewrite      |
                   Rewrite

Rewrite        ::= Formula == Formula
```

An example of a rewrite rule is:-

 ©1996 *B-Core (UK) Ltd.*

```
 not(s = {})
=>
 max({max(s),x}) == max(s\/{x})
```

This rule states, in effect, that, if `not(s = {})` can be proved, any sub-formula of the current goal that has the form `max({max(s),x})` can be rewritten as `max(s\/{x})`.

The Jokers `s` and `x` can be replaced by any formula, so that applying this rule to the goal

```
 max({max({0}),new}) == max({0}\/{new})
```

has the effect of applying the rule

```
 not({0} = {})
=>
 max({max({0}),new}) == max({0}\/{new})
```

Rewrite rules are used for Backwards Inference, and are applied as follows:-

1. The left-hand side of the Rewrite is matched to any sub-formula of the current goal by substituting a formula for each Joker which makes the Consequent identical to the goal. If no such match can be found, the rule cannot be applied.
2. If the match is successful, the same substitution of formulae for Jokers is applied to the right-hand side of the Rewrite, and to Antecedents (if any).
3. The current goal is replaced by a set of goals which comprises:-
   - the new goal resulting from applying the Rewrite to the matched sub-formula of the current goal; and
   - the new Antecedents (if any).

Note that a Rewrite Rule need not have Antecedents.

6.1.4 Forward Rules

A Forward Rule has the following syntax:-

```
Forward_Rule ::= Antecedents => Consequent
```

An example of a Forward Rule is :-

 ©1996 *B-Core (UK) Ltd.*

```
not(a:T) &
a:(S\/T)
=>
a:S
```

This rule states, in effect, that, if it is known that `not(a:T)` and that `a:(S\/T)`, then it is also known that `a:S`.

The Jokers `a`, `S` and `T` can be replaced by any formula, so that applying this rule to the hypotheses

```
not(percy:STAFF) &
percy:(CUSTOMER\/STAFF)
```

has the effect of applying the rule

```
not(percy:STAFF) &
percy:(CUSTOMER\/STAFF)
=>
percy:CUSTOMER
```

Forward rules are used for Forwards Inference, which is used every time a new hypothesis is introduced into the set of hypotheses.

Forwards rules are applied as follows:-

1. The Antecedents are matched to the current set of hypotheses by substituting a formula for each Joker which makes each Antecedent identical to one of the hypotheses. If no complete match can be found, the rule cannot be applied.
2. If the match is successful, the same substitution of formulae for Jokers is applied to the Consequent.
3. The new consequent is added to the current set of hypotheses.

Note that a Forwards Rule must always have Antecedents.

6.1.5 Backward Rules

Backwards Rules are Simple Rules or Rewrite Rules.

 ©1996 *B-Core (UK) Ltd.*

6.1.6 Pattern Matching

The presence of jokers in formulae allow pattern matching to take place. A formula `f` matches a formula `g` if a substitution `S` of formulae for the jokers of `f` can be found such that `[S]f` is identical to `g`.

For example, matching the formula

```
age(p):T
```

to the formula

```
age(percy) : 0..120
```

is achieved using the substitution

```
p , T := percy , (0..120)
```

Special care must be taken with the matching to comma-separated lists. The comma operator is treated just like other left-associative infix operators. So the formula `{p}` can match the formula `{1,2,3}` with substitution

```
p := 1,2,3
```

whereas `{P,p}` would match as

```
P , p := (1,2) , 3
```

6.1.7 Guards

Guards are a special kind of Antecedent for backwards rules that are evaluated in a different way during proof construction. The proof mechanism prevents rules from being applied if the guards do not evaluate to true.

Guards also provide a means of making inference rules more generic. Their syntax is as follows:

```
Guards      ::= Guards & Guard                                          |
                Guard
```

 ©1996 *B-Core (UK) Ltd.*

```
Guard      ::= "binhyp" ( Formula )                                    |
               "bnum" ( Formula )                                      |
               "bident" ( Formula )                                    |
               "bstring" ( Formula )                                   |
               "btest" ( Number Comparison Number )                    |
               "bsearch" ( Formula , ( Formula_List ) , Formula )      |
               "breade" ( Formula , Formula_List , Formula )

Comparison ::=  <  |  <=  |  >  |  >=  |  =  |  /=
```

In this context, a `Number` is any Natural number less than 2^32. A `Formula_List` is comma (or in some cases "`\&`", "`or`" or "`;`") separated list of Formulae.

A brief explanation of these guards is as follows:-

`binhyp(x)`
: succeeds if there is a hypothesis which matches `x`. For example, application of the following the simple rule

```
binhyp(f:S-->T) &
x:S
=>
f(x) : T
```

to the formula

```
age(pers) : 0..120
```

in the presence of the hypothesis

```
age:PERSON-->0..120
```

behaves as if the following rule is applied:-

```
age:PERSON-->0..120 &
pers:PERSON
=>
age(pers) : 0..120
```

`btest( x > y )`
: succeeds if `x` and `y` are valid numbers, and `x` is greater than `y` (and similarly for the other comparison operators). For example:-

```
btest(a/=b)
=>
not(a:{b})
```

 ©1996 *B-Core (UK) Ltd.*

`bsearch(x,(l),r)`

succeeds if the formula `x` matches a formula in the list `l`, and `r` is the list `l` with the matched formula removed. The list can be separated by commas, "`\&`" or "`or`".

For example, a typical use of this guard is as follows:-

```
bsearch((x < y),P,Q) &
[x,y := 1,2]P
=>
#(x,y).P
```

This rule can be used to attempt to prove an existential quantifier in which `P` contains the constraint x by selecting specific values for `x` and `y`, namely `1` and `2`.

The antecedent `[x,y := 1,2]P` is a predicate identical to `P` with all free occurrences of `x` and `y` replaced by `1` and `2` respectively. So applying this rule to the goal

```
#(a,b).( a:NAT & b:NAT & a < b )
```

has the same effect as applying the rule

```
1:NAT & 2:NAT & 1 < 2
=>
#(a,b).( a:NAT & b:NAT & a &lt; b )
```

Careful attention is given, of course, the renaming of variables to ensure that clashes with free variables are avoided.

`breade(f,l,x)`

displays the list of formulae `l` on the screen according to the format given in the quoted string `f`, and succeeds if `x` matches the response typed by the user.

The format string `f` works on the same kind of principle as `printf` function in C. The `%` character is used to mark the position of formulas in the output, and various escape characters are allowed, such as follows:-

- `\n` line feed
- `\t` tabulation
- `\E` escape
- `\B` bell
- `\"` double quote
- `\\` back slash
- `\%` percent

This guard can be used to assist in the selection of values in the proof of existential quantification, or quite simply to trace the application of user-defined rules.

 ©1996 *B-Core (UK) Ltd.*

For example, another way of treating the existential quantification above is by providing the rule:-

```
breade("\n -- Choose values for %,% which satisfy % : ",x,y,P,(s,t)) &
[x,y := s,t]Q
=>
#(x,y).P
```

Attempted application of this rule to the goal

```
#(a,b).( a:NAT & b:NAT & a<b )
```

would result in the following message being displayed on the screen:-

```
-- Choose values for a,b which satisfy (a:NAT & b:NAT & a<b) :
```

to which the user may respond by typing `1,2` for the same effect of the `bsearch` example above.

The interactive tracing of attempted rule applications can be achieved by including antecedents of the following form:-

```
breade("\n *** APPLYING MY SPECIAL RULE TO % : ",(#(x,y).P),v) &
bsearch((x < y),P,Q) &
[x,y := 1,2]Q
=>
#(x,y).P
```

`bident(x)`
: succeeds if `x` is a valid identifier (a string of length greater than 1 comprising upper or lower-case letters, digits and the underscore character, and containing at least one letter).

`bnum(x)`
: succeeds if `x` is a valid number (a Natural number less than 2^32).

`bstring(x)`
: succeeds if `x` is a valid quoted string (any character string enclosed in double quotes).

An example of the last of these is:-

```
bident(b)
=>
{a} \/ {b} == {a,b}
```

Note the subtle difference in the syntax for `bsearch` and `breade`. The second argument of `bsearch` must be a list enclosed in brackets, whereas the list in `breade` does not, giving rise, in effect, to a variable number of arguments (rather like the `printf` function in C). The `x` thus refers, in fact, to the last argument.

 ©1996 *B-Core (UK) Ltd.*

6.1.8 Theories

A Theory is a container for inference rules. A forward theory contains forward rules, and a backward theory contains backwards rules.

```
Theory      ::= THEORY Theory_Name IS Rule_List END

Theory_Name ::= Identifier

Rule_List   ::= Rule_List ; Inference_Rule  |
                Inference_Rule
```

In the Proof Method files presented by the user to the InterProver, backwards rules and forwards rules are presented in two theories, usually called `UsersTheory` and `FwdUsersTheory` respectively. There is no reason, however, why users should not present any number of theories with others names, as long as the appropriate tactics are also amended.

The order of rules in a theory is significant. (See Ordering Rules in Theories below for a discussion of this.)

6.1.9 Tactics

B-TL provides a very simple tactic language which guides proof construction. The language describes sequencing and iteration of theory applications, and names some basic atomic tactics. The syntax is as follows:-

```
Tactic        ::= Atomic_Tactic    |
                  Tactic ; Tactic  |
                  Tactic~

Atomic_Tactic ::= DED  |  GEN  |  SUB  |  HYP  |  FEQL  |  Theory_Name
```

The semicolon indicated sequencing of actions, and the tilda indicates repetition. The following is an example of a tactic:-

```
( Theory1 ; DED ) ~ ; Theory2~
```

which says, in effect:-

1. First attempt to apply any rule from theory `Theory1`, then apply the special tactic `DED`.
2. Repeat 1 until no more rules apply.

 ©1996 *B-Core (UK) Ltd.*

3. Then repeatedly apply any rules from `Theory2` until no more apply.

In the Proof Method files used by the InterProver, two tactics are named in the `USETAC` clause (see section on InterProver). The first is the tactic to be used for backwards inference, and the second is for forwards inference.

By default, these two tactics are presented as:-

```
(UsersTheory~),

(FwdUsersTheory~/*;FEQL~*/)
```

and the user is expected to present backwards rules in a theory called `UsersTheory` and forwards rules in `FwdUsersTheory`.

Note that the application of the special tactic FEQL is commented out by the annotation marks `/*` and `*/`. It is placed there to remind users that a frequently desired modification of the forward tactic is to include the "forwards equality" tactic. In some applications, this modification helps to push proofs through, often without providing additional rules. It is left out of the default for efficiency reasons.

The special tactics are described as follows:-

`DED`
: The deduction tactic, `DED`, acts on goals of the form

  ```
  H1 & H2 & ... Hn  =>  G
  ```

  by placing the formulae `H1`, `H2` ... `Hn` into the current set of hypotheses, and making `G` the current goal.

`GEN`
: The generalisation tactic, `GEN`, acts on goals of the form

  ```
  !v.G
  ```

  by renaming all the variable `v` in `G` to avoid free occurrences in the current set of hypotheses, and making the renamed `G` the new goal.

`SUB`
: The substitution tactic, `SUB`, acts on goals with contain sub-formulae of the form

  ```
  [v:=F]G
  ```

  by performing the substitution on `G`, and making the rewritten goal the new goal.

`HYP`
: The "hypothesis" tactic, `HYP`, can act in a number of ways. Suppose the current goal is `Q`:-

 ©1996 *B-Core (UK) Ltd.*

- When there is a hypothesis in the current set of the form

  ```
  P1 & P2 ... Pn => Q
  ```

  the current goal `Q` is replaced by `P1`, `P2` ... and `Pn`.
- When there is a hypothesis of the form

  ```
  !v.G
  ```

  where `G` is identical to `Q` for some value of `v`, then the current goal `Q` is discharged.

`FEQL`

The "forward equality" tactic, `FEQL` is a forward tactic which generates new hypotheses from the set of current hypotheses by calculating the transitive closure of equality.

For example, if the set of current hypotheses contains

```
newperson = percy
newage = 23
age +> {newperson |-> newage} <: PERSON --> AGE
```

then the FEQL tactic would have the effect of expanding the set to

```
newperson = percy
newage = 23
age +> {newperson |-> newage} <: PERSON --> AGE
age +> {percy |-> 23} <: PERSON --> AGE
```

6.1.10 Proof Construction

Proofs are constructed in the context of:-

- A set of goals, one of which is the current goal.
- A set of current hypotheses.
- A normal tactic, with a current position.
- A forwards tactic, with a current position.
- A set of theories.

At the start of proof:-

- A single goal exists in the set of goals, which is selected as the current goal.
- The set of current hypotheses is empty.

 ©1996 *B-Core (UK) Ltd.*

- Normal and forwards tactic positions are pointing at the first theory or special tactic.

As proof proceeds, the set of current hypotheses, the set of goals, the choice of current goal, and the tactic positions are manipulated as follows:-

1. An attempt is made to find a rule from the theory named at the current current normal tactic position which matches the current goal.
2. If a matching backwards rule is found, it is applied, which may result in the addition of hypotheses to the set of current hypotheses, and to manipulation of the set of goals, as described in the section describing backwards inference.
3. If the set of goals is now empty, proof is concluded with success.
4. If new hypotheses are introduced, forward inference is applied as described below.
5. The normal tactic position is incremented, taking repetition into account.
6. If the end of the normal tactic is reached, proof terminates unsuccessfully.
7. A new current goal is selected (usually a child of the last current goal).
8. Proof construction is repeated from 1.

The forward inference process referenced in step 4 above is as follows:-

1. An attempt is made to find a rule from the theory (or special tactic) named at the current forwards tactic position which matches formulae from the set of current hypotheses.
2. If a matching forwards rule is found, it is applied, which may result in the addition of hypotheses to the set of current hypotheses.
3. The forwards tactic position is incremented, taking repetition into account.
4. Forward inference is repeated from 1.
5. When the end of the forwards tactic is reached, forwards inference terminates.

 ©1996 *B-Core (UK) Ltd.*

6.1.11 Guidelines for Proof

The following notes are intended to guide the B-Toolkit in constructing theories for use with the InterProver.

6.1.12 Ordering Rules in Theories

When applying rules from a theory, rules are matched in strict biblical order: the last first, and the first last. This allows a further aspect of control over which rules are applied: a rule is applied only if all other rules below it in the theory have not matched.

For example, consider the following theory:-

```
THEORY max IS
    max({p}) == p
;
    max({max(P)}\/Q) == max(P\/Q)
;
    max({P,p}) == maxel({P,p},0)
;
    maxel({p},q) == p
;
    btest(q>p)
   =>
    maxel({p},q) == p
;
    maxel({P,p},q) == maxel({P},p)
;
    btest(q>p)
   =>
    maxel({P,p},q) == maxel({P},q)
END
```

The last rule is attempted first. This rule is guarded by a `btest`. If this fails, then the next rule above is attempted.

The rewrite in the next rule has an identical left-hand side, so the rule will match exactly when the bottom rule does. This time, however, there is no guard, but we know from the failure of the rule below that it will only apply when `q<=p`.

Together, these last two rules cover all cases when there are at least two elements in the set in the first argument of `maxel`.

The next rule up has a rewrite with a different (more general) left-hand side. The formula `{p}` can match any set, including those of the form `{P,p}` (see Pattern Matching). But

 ©1996 *B-Core (UK) Ltd.*

we know that, since the previous rules did not match, the only pattern that remains to be matched is the single set, in which matches a single element, and not a comma-separated list. A similar argument applies to the top rule.

6.1.13 Using Guards in Rewrite Rules

The use of guards is strongly advised for rewrite rules. This is because there is limited backtracking through rewrite rules: once they are applied, they cannot be unapplied through backtracking. There is a need, therefore, to have careful control over the application of rewrite rules.

The first control over the application of a rewrite rule is through pattern matching: the rule will only apply if the shape of the left-hand side of the rewrite matches a sub-formula of the current goal.

A second level of control is by judicious ordering of rewrite rules in the theory. (See Ordering Rules in Theories above.)

A third level of control is by using guards in the antecedent of the rule: if guards evaluate to false, the rule is not applied.

A better version of the rewrite rule example given above is thus:-

```
 binhyp(not(s = {}))
=>
 max({max(s),x}) == max(s\/{x})
```

since this only allows rewriting when it is known from the current set of hypotheses that `not(s = {})`.

6.1.14 Backwards versus Forwards Inference

Backwards inference is goal-oriented, in that rules matched against the current goal. It is therefore inherently more efficient than forwards inference, which is not goal-directed, since it blindly propagates knowledge from the hypotheses.

It is therefore recommended that simple rules and rewrite rules are used in preference to forwards rules where ever possible.

6.1.15 Supplying Rules for Animation

For the purposes of Animation of AMN specifications in the B-Toolkit, a theory file can be supplied containing only guarded or unguarded rewrite rules. Other kinds of rules would

 ©1996 *B-Core (UK) Ltd.*

interfere with the animation mechanism. (See section on the Animator entitled Providing More Rewrite Rules.)

6.1.16 Running Out of Memory

When conducting really large proofs, it is possible for the B-Platform to run out of memory. If this occurs, an error message is displayed in the AutoProver or InterProver panel, and the proof session is aborted.

Instructions for recompiling the B-Platform with more space are given in the Installation Guide , section 1.3.4.

 ©1996 *B-Core (UK) Ltd.*

6.2 The POGenerator (pog)

The POGenerator (pog) is invoked from the Main Environment, and is applicable to all non-library, non-generated specifications refinements and implementations which are not currently generated; the Analyser is invoked first, where necessary.

The tool automatically generates all the proof obligations which are required to be discharged in order to guarantee the internal consistency of a specification or the correctness of a refinement with respect to its abstraction. The POGenerator provides the same options as the Analyser when a construct being currently edited differs from that of the committed version.

The proof obligations are generated according to the correctness criteria which are required to hold within the B-Method. Thus, for example, the criteria require that an Abstract Machine initialisation must establish the invariant, and that each operation re-establish the invariant. Obligations are also generated to maintain the correct relationships between a machine and its refinements. (Full details of the proof obligations are given in the B-Technology Technical Overview.)

The following three-letter acronyms appear in B-Toolkit proof obligations:

- `cst`
  the constraints of the construct (the predicates of the CONSTRAINTS clause)
- `ctx`
  the context of the construct (the predicates of the PROPERTIES clause of the construct together with those of subordinate constructs)
- `inv`
  the invariant of the construct (the predicates of the INVARIANT clause, together with abstract/concrete equalities of any algorithmically-refined variables)
- `asn`
  the assertions of the construct (the predicates of the ASSERTIONS clause)
- `pre`
  the precondition of the operation (the the predicates of the operation's PRE clause)

The generated proof obligations are stored in the environment directory POB, but some very simple obligations are discharged by the tool and do not appear in the proof obligation file.

If an error is detected during generation of the proof obligations then the Mini-remake option is offered.

 ©1996 *B-Core (UK) Ltd.*

The current number of outstanding proof obligations and total number of proof obligations (where applicable) is displayed for each construct in the Constructs Area in the Provers Environment window under 'pob' and 'tot' respectively.

©1996 *B-Core (UK) Ltd.*

6.3 The AutoProver (apr)

The AutoProver (apr) is invoked from the Provers Environment and is applicable to all machines, refinements, and implementations, which are pogenerated, and currently have undischarged proof obligations.

It is a tool which attempts to discharge the proof obligations generated by the POGenerator using in-built tactics and a standard library of mathematical laws.

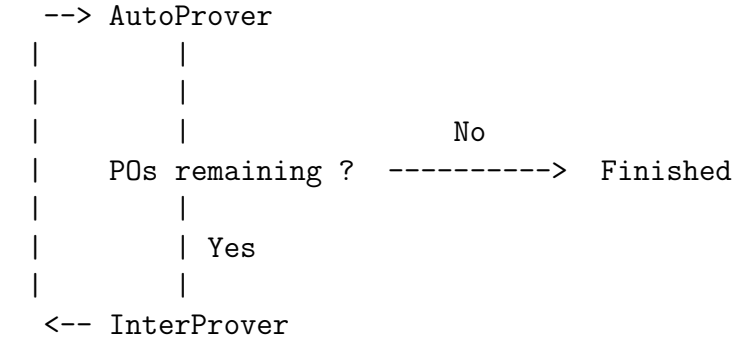
Some obligations may not be proved by the AutoProver. This may be because an obligation is not provable (i.e. there is a mistake in a specification/refinement/implementation), or because the AutoProver's tactics or standard library are inadequate for the proof. In order to supplement the library with additional mathematical laws and/or different tactics, an InterProver session is required.

It is stressed that the InterProver is a BROWSING tool (no 'proofs' are committed to the system) which facilitates the inspection of the points at which the AutoProver failed in its attempts to discharge a proof obligation. It enables you to supplement the AutoProver's standard in-built library of mathematical laws with additional laws of your own through the user's PROOFMETHOD file.

Thus different 'Levels' of AutoProof are possible, allowing the user to concentrate on different proof obligations at each one, if required; at each Level, the following may be set through the PROOFMETHOD:-

- the proof obligations to attempt
- the level of forward reasoning depth that the AutoProver should use
- the user-supplied theory
- the user-supplied tactic

The AutoProver-InterProver cycle has the following structure:



 ©1996 *B-Core (UK) Ltd.*

At each level, the AutoProver's input is a proof obligation file and PROOFMETHOD file (for example at Level 1 'fifi.mch.1.po' and 'fifi.mch.1.pmd'); its output is an updated proof obligation file 'fifi.mch.2.po' and a template proof obligation file 'fifi.mch.2.pmd' (if the latter does not already exist in the PMD, and there remain undischarged proof obligations). This is the used in investigation of the undischarged proof obligations through the next InterProver, session, and subsequent use by the AutoProver at the next level.

During the AutoProof the status of each obligation is represented by a single character as follows:

```
p       indicating it has already been discharged
        at a previous proof level

.       indicating it is not being attempted at this level
        (see the ON clause of the PROOFMETHOD)

+       indicating that it was successfully discharged at this level

-       indicating that it was attempted but not discharged at this
```

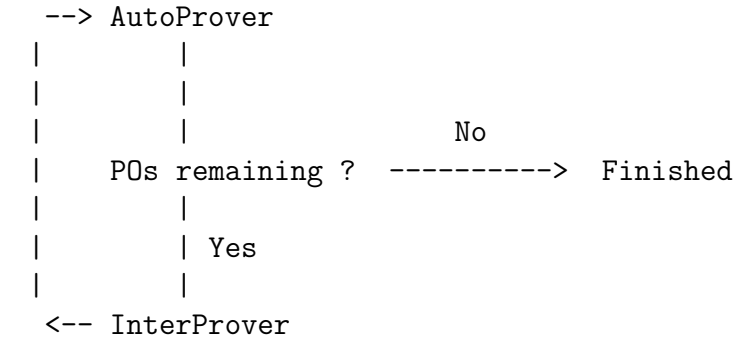
The current number of outstanding proof obligations and total number of proof obligations (where applicable) is displayed for each construct in the Constructs Area of the Provers Environment panel under 'pob' and 'tot' respectively.

 ©1996 *B-Core (UK) Ltd.*

6.4 The InterProver (ipr)

The InterProver (ipr) is invoked from the Provers Environment and is applicable to all machines, refinements and implementations which are pogenerated and currently have undischarged proof obligations.

If the AutoProver fails to discharge all the proof obligations for a given construct then proofs of the remaining obligations may be attempted by entering the AutoProver-InterProver cycle which has the following structure:



The InterProver allows you to investigate why the AutoProver has failed to discharge a proof obligation. It is stressed that it is a BROWSING tool (no 'proofs' are committed to the system) which facilitates the inspection of the points at which the AutoProver failed in its attempts to discharge a proof obligation. It enables you to supplement the AutoProver's standard in-built library of mathematical laws with additional laws of your own through the user's PROOFMETHOD file.

InterProving comprises two distinct phases:

1. Finding an appropriate point in the INCOMPLETE proof where you think you can provide the mathematical insight to enable the proof to proceed. This is achieved by creating one or more lemmas. This phase uses only the in-built tactics and theory of the AutoProver

2. Providing the necessary mathematics in order to COMPLETE the proof. This is achieved by discharging the lemmas This phase uses the in-built tactics and theory of the AutoProver supplemented by user-provided theory and tactics.

Although the InterProver actually constructs proofs, its browsing nature means that such proofs are not committed to the system. The idea is that, having demonstrated to yourself that the mathematical insight that you have provided is sufficient, you then leave the InterProver and invoke the AutoProver, which constructs the next level of proof using your PROOFMETHOD file, which you have already demonstrated is adequate.

 ©1996 *B-Core (UK) Ltd.*

The InterProver is invoked from the Provers Environment by first selecting a construct and then activating the InterProver (ipr). You are then presented with a selection-list of item names (e.g operation names). For each item the number of (unproved) proof obligations associated with this item is given.

An editor window is also presented which contains the appropriate PROOFMETHOD file, if one exists, or a template proof method file otherwise. This template can be edited to provide the provers with user-supplied laws of mathematics and proof tactics, as described below. (Each prover cycle (InterProver - AutoProver) creates a new "proof level" with its own PROOFMETHOD file stored in the environment subdirectory PMD).

On selection of one of the items (e.g operation), another selection list of the proof obligations outstanding for that item is presented. Once one of these proof obligations has been selected, that proof obligation is displayed as the Current Goal for the InterProver and you are presented with a menu of options for driving the InterProver in producing an INCOMPLETE proof.

6.4.1 The First Phase - Creation of Lemmas

The options are initially:

- Run Prover
- Create Lemma

The 'Run Prover' option uses the rules in the standard library in an attempt prove the Current Goal; the InterProver uses the same standard library as the AutoProver.

Selecting the 'Run Prover' may result in either the goal being discharged or in the InterProver encountering a goal for which no standard rules apply, when it stops and presents the options:

- Backtrack
- Show Hypotheses
- Create Lemma

The 'Create Lemma' option discharges the Current Goal through the creation of a Lemma; this should be employed if you believe that the Current Goal is provable, and that you can easily supply some additional laws of mathematics which prove that lemma. (At this point you might provide these additional laws by entering them as rules in the UsersTheory in the provided PROOFMETHOD file, although they won't be used until the second phase).

 ©1996 *B-Core (UK) Ltd.*

The hypotheses associated with the Current Goal can be displayed through 'Show Hypotheses' at any time. This might assist you in assessing whether a goal is provable. The current set of hypotheses is displayed inside an editor or other tool specified by the environment variable `BHYPEDIT`.

The 'Backtrack' option provides a backtracking mechanism, and should be selected if you think that the Proof Obligation is provable and if you think that the Current Goal is too difficult to justify. Selecting run will backtrack to an intermediate goal and you will at this stage be informed by the InterProver of how (and why) the goal, from which you are backtracking, were produced.

The intermediate goals presented may be altered by setting the 'InterProver Reasoning Depth' to the appropriate level of forward reasoning:

- InterProver Depth 0
- InterProver Depth 1
- InterProver Depth 2

(The effects of these settings are the same as those described for the REASONING_DEPTH clause of the PROOFMETHOD)

In order to use the InterProver to produce an INCOMPLETE proof for a Proof Obligation you will have to create a lemma at some point of your choosing, either for a final goal or an intermediate goal. If you do not create any lemmas then repeated use of the run option will eventually return you to the Original Proof Obligation.

If you need to investigate the proof attempt again you can repeat the run. Note that a single proof obligation may give rise to several lemmas. During this first phase, the PROOFMETHOD is not consulted, and so the intermediate goals that are presented stem from the AutoProver's in-built tactics and theory. It is possible to change the intermediate goals only through the the setting of the 'InterProver Reasoning Depth'.

The display shows the Original Proof Obligation and the Current Goal (on which the AutoProver can currently make no progress):

See figure 14.

A combination of the Current Hypotheses and the Current Goal provides the information necessary to enable the user to decide whether or not to create a lemma, and assist in providing the mathematics to be used in the discharge of that lemma in the second phase (all rules are written in the B-Theory Language of the BPlatform):

See figure 15.

Having created one or more lemmas (which are stored in a selectable item called 'Lemmas') you can attempt to prove them completely by returning to the 'Select Theory' option -

 ©1996 *B-Core (UK) Ltd.*

Figure 14: Lemma Creation

```
max(s\/{t}\/u) == max({max(s\/u),t})
```

Figure 15: PROOFMETHOD Rule

 ©1996 *B-Core (UK) Ltd.*

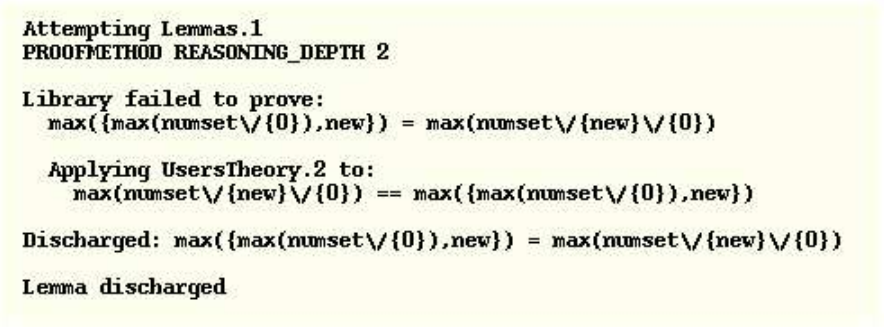


Figure 16: Lemma Proof

through 'Cancel' - and selecting the Lemmas item. Note that when a proof obligation is discharged by the creation of lemmas, you are automatically returned to the 'Select Theory' option.

6.4.2 The Second Phase - Proof of Lemmas

This second phase - attempting the proof of a lemma - now simulates the action of the AutoProver which is dictated by the PROOFMETHOD file, and so, after selection of a particular Lemma, that file is automatically loaded (provided that it parses!), and you the are presented with the choice:

- Run Prover
- Show Hypotheses
- Reload PROOFMETHOD

Note that there is now no 'Backtracking' option, and that the 'InterProver Reasoning Depth' option is no longer available (this is now a simulation of the AutoProver).

On selection of 'Run Prover' your PROOFMETHOD file is used by the system to attempt a COMPLETE proof. A tracing mechanism indicates which of your rules are applied, and to which goals they are applied, in order to assist in understanding the proof process.

See figure 16.

If the attempt is unsuccessful, the tracing information provided should be sufficient to indicate the way in which the PROOFMETHOD should be edited, and having done this it may be loaded using the 'Reload' option when the previous PROOFMETHOD is removed and the current one loaded.

The 'Show Hypotheses' option will show the hypothesis associated with the final goal produced during a failed attempt to prove a lemma. Note that when the PROOFMETHOD is reloaded all previously proved lemmas re-appear (since they may no longer be provable once the proof method has been edited!).

You can follow this cycle as many times as you want until either the lemma is proved, or you wish to abort the attempt - through 'Cancel'.

6.4.3 The PROOFMETHOD file

The PROOFMETHOD file resides in the environment subdirectory PMD and is automatically presented for editing during InterProof. It may also be edited through the 'pmd' button of the Provers Environment.

The syntax of a proof method is:

```
PROOFMETHOD
    name
IS
   USETAC tactic
    [ REASONING_DEPTH  n1 ]
    [ ON lemma_list ]
   END

  [ &

   THEORY th1 IS rules1 END

   &
    ...
   &

   THEORY thN IS rulesN END ]
END

[
DEFINITIONS
    rewrite_rule [ ; rewrite_rule ]
INCLUSIONS
    incl_file [ , incl_file ]
]
```

Theories `th1,..,thN` contain the user-supplied rules: `rulesq,..,rulesN`.

 ©1996 *B-Core (UK) Ltd.*

The optional DEFINITIONS clause allows macro definitions in the form of rewrite rules to be applied to the user-supplied Theories before use. A DEFINITIONS clause comprising '?' is equivalent to an empty clause.

The optional INCLUSIONS clause allows other PROOFMETHOD files to be loaded (and, of course, the process is recursive). For such included files, however, the USETAC, REASONING_DEPTH and ON clauses are ignored; those settings are taken from the main PROOFMETHOD file.

Each inclusion should be the name of a PROOFMETHOD file enclosed in double quotes, and for proof levels other than zero, the default file produced by the system contains a single INCLUSIONS entry comprising the previous-level PROOFMETHOD file. An INCLUSIONS clause comprising '?' is equivalent to an empty clause.

The optional REASONING_DEPTH clause enables the provers to work with different sets of 'forward' rules:-

- when `n1` = 0
  only a simple set of rules for recognising mathematical contradictions are used, and furthermore the antecedent of the proof-obligations is not inspected during proofs;
- when `n1` = 1
  the antecedent of the proof-obligations is used and a simple set of rules for producing further consequences from the specific antecedent are used;
- when `n1` = 2
  an extended set of rules for producing consequences are used.

NOTE: the performance of the provers is affected by this setting.

The optional ON clause enables the AutoProver to subsequently attempt to prove a subset of undischarged proof obligations; 'lemma_list' is a comma-separated list containing either:

- name
  (e.g an operation name) indicating that all outstanding obligations of that name should be attempted
- name.n
  indicating that only the nth obligation of name should be attempted
- name.(n,m,..)
  indicating that name.n, name.m, name... should be attempted.

If the ON clause is omitted, the AutoProver will use the supplied tactics to attempt all unproved proof obligations.

 ©1996 *B-Core (UK) Ltd.*

The tactic is a standard B-TL tactic, i.e. a pair `(b,f)` consisting of a normal tactic `b` and a forward tactic `f`. (See section called "B-Platform" for a discussion of proof and tactics.)

In attempting a COMPLETE proof, the InterProver and the AutoProver will follow their own in-built tactics until a goal is (or goals are) generated which the Provers cannot match. At this point, the supplied user tactics are adopted.

Each generated PROOFMETHOD file is provided with a default tactic, which may be edited by the user. The default normal tactic is

```
(UsersTheory~)
```

which is the theory (provided through the PROOFMETHOD) that the provers tactic switches to after the in-built tactic is tried. If progress is made through one of the rules in 'UsersTheory', the tactic switches back to the in-built tactic until no progress can be made; again the tactic switches to that of the PROOFMETHOD file and the process repeats until either the proof is complete, or no further progress can be made. The in-built tactic also incorporates a backtracking mechanism so that all possible proofs are attempted before the process is deemed to have been unsuccessful.

The default forward tactic is

```
(FwdUsersTheory~/*;FEQL~*/)
```

Here, FwdUsersTheory is employed (together with FEQL if not commented out) before the library's standard forward tactic.

You should always edit the PROOFMETHOD file using the InterProver editor, since this editor is the means of propagating the effects of changes in the proof method file to the configuration tools (i.e. recording the 'level' in the prover cycle).

Having proved all (or some) of the outstanding proof obligations the cycle is completed by returning to the top level of the prover environment and invoking the AutoProver; it uses the proof method file just edited to discharge more proof obligations and, if proof obligations remain, a new level is generated, enabling the cycle to be repeated.

 ©1996 *B-Core (UK) Ltd.*

6.5 The Proof Printer (ppf)

The ProofPrinter (ppf) is invoked from the Provers Environment, and is applicable to all machines, refinements and implementations which are pogenerated/ autoproved. Invoking the tool results in a selection showing those levels for the construct which have not been printed.

A 'prf' construct containing either statements of proof obligations (in the case of Level 0 constructs produced by the POGenarator), or proof listings (for Levels greater than 0 produced by the AutoProver) results.

Proof constructs are displayed in both the Provers and Documents Environments.

In the case of proofs, Laws (either those provided by the B-Toolkit Library, or those provided by the user through a PROOFMETHOD) used in proofs contained in that '.prf' construct are presented at the end of the construct; note that the scope of the numbering of such Laws is the file in which they appear.

Each proof conforms to a B-formula and so may be machine-processed by other proof checkers, if required, to check the validity of the proof.

Each proof construct may be marked up and included in a DOCUMENT if required; this is accomplished in the Documents Environment.

Although the formatting is completely automatic, it is sometimes necessary to insert line-breaks or pagebreaks.

A linebreak - the LaTeX '\\' - may be inserted immediately following an infix symbol. For example, following an ampersand '`&`' or implication '`=>`' (to break predicates) or a comma ',' (to break justification lists).

A pagebreak - the LaTeX '`\newpage`' - may be inserted immediately following the semicolon '`;`' which separates the proofs/Laws.

No other editing of proof constructs is permitted!

 ©1996 *B-Core (UK) Ltd.*

6.6 ShowUnproved (sup)

This option is invoked from the Provers Environment and is applicable to all machines, refinements, and implementations, which currently have undischarged proof obligations; a list of such obligations is displayed in the Display Area in the form 'operation.number'.

©1996 *B-Core (UK) Ltd.*

6.7 ResetLevel (rsl)

The results of an automatic proof session may be removed using the ResetLevel (rsl) option in the Provers Environment, when a proof level preceding the current one may be substituted. The option is available only when at least one autoproof has currently been performed. Note that using this facility will cause proof levels to be completely removed from configuration, and a subsequent remake will build only to the newly-set level.

©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

B-TOOLKIT USERS' MANUAL

SECTION 7

Generators

©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

7 Generators

7.1 The Base Generators (gbo) (gbm)

The Base Generator tools are invoked from the Generators Environment and take as input a base construct file, with file extension '.bse' (giving a description, in the BASE language, of a SYSTEM). The output comprises a specification and matching implementation of the system described in the base. The implementation is built on the SLIB library machines.

These tools provide tailored, purpose-built specifications and implementations of objects built as structured data with a rich set of operations to manipulate that data. The data is described in a declarative manner in the '.bse' file, and the operations produced may be filtered through editing (manually, or automatically through an optional SUPPORTS clause) a generated '.ops' file.

An input of, for example, 'fifi.bse' would produce an output, to the CFG directory, of four configured Abstract Machine source files `fifi.mch`, `fifiCtx.mch`, `fifiI.imp` and `fifiCtxI.imp`; it also outputs related analysed and type files to the ANL and TYP directories. The Ctx machines provide contextual information regarding sets, constants and properties, and are not present if the base description comprises a single GLOBAL base.

Abstract Machines produced by the tool are standard specifications which can be imported or included into other machines. The output source file provides the state and operations available on the state. Since these are generated machines, you can assume that they are syntactically correct, well-typed and internally consistent - hence no proof obligations are generated.

The generation of these machines takes place in two stages: first a '.ops' construct is generated (through the 'gbo' directive); this comprises a file of operation headers which will dictate which operations will subsequently be produced by the 'gbm' directive.

The '.ops' file may be edited to reduce the number of operations produced in line with the user's requirements (it is recommended that operations not required are commented out rather than removed). This '.ops' construct is dependent on the '.bse' construct, and all generated machines are dependent on the '.ops' construct.

The general form of a system description is given in a BNF-like syntax as follows:
the syntax `exp1 | exp2` indicates `exp1` or `exp2` (choice),
`<< exp >>` indicates zero or more occurrences of `exp` (repetition) and
`[| exp |]` indicates optionality.

```
system_description ::=  SYSTEM system_name
                        [|
                          SUPPORTS imp_name << , imp_name >>
                        |]
```

 ©1996 *B-Core (UK) Ltd.*

```
                        IS system_body END

system_name         ::=  Identifier

imp_name            ::=  Identifier

system_body         ::=  GLOBAL global_dec_list END << ; base_body >> |
                         base_body << ; base_body >>

base_body           ::=  BASE base_name base_clause END |

base_name           ::=  Identifier

base_clause         ::=  MANDATORY mand_dec_list |
                         OPTIONAL opt_dec_list |
                         MANDATORY mand_dec_list OPTIONAL opt_dec_list

opt_dec_list        ::=  opt_dec << ; opt_dec >>

mand_dec_list       ::=  mand_dec << ; mand_dec >>

opt_dec             ::=  Identifier: STRING[dim] |
                         Identifier: SEQ(Identifier)[dim] |
                         Identifier: SET(Identifier)[dim] |
                         Identifier: Identifier

mand_dec            ::=  Identifier: STRING[dim] |
                         Identifier: FSTRING[dim] |
                         Identifier: SEQ(Identifier)[dim] |
                         Identifier: SET(Identifier)[dim] |
                         Identifier: Identifier

global_dec_list     ::=  global_dec << ; global_dec >>

global_dec          ::=  Identifier: STRING[dim] |
                         Identifier: FSTRING[dim] |
                         Identifier: SEQ(Identifier)[dim] |
                         Identifier: SET(Identifier)[dim]
```

 ©1996 *B-Core (UK) Ltd.*

```
dim                ::=  Bnumber
```

where Identifier is a B Identifier.

An example system description is:

```
SYSTEM
   example
SUPPORTS
   fifiI, mimiI
IS

  GLOBAL
    gg2: SEQ(base1)[10];
    gg3: SET(base1)[15];
    gg4: FSTRING[8]
  END;

  BASE
    base1
  MANDATORY
    tt1: base2;
    tt2: SEQ(base2)[5];
    tt3: SET(base1)[2];
    tt4: STRING[10]
  OPTIONAL
    pp1: NAT;
    pp2: SEQ(base1)[10];
    pp3: SET(PERSON)[25];
    pp4: STRING[12]
  END;

  BASE
     base2
  OPTIONAL
    pp5: PERSON
  END

END
```

This stipulates a module (state machine) consisting of three simple data structures `gg2`, `gg3` and `gg4`, and two complex data structures `base1` and `base2`. Complex data structures may have mandatory (`MANDATORY`) and optional (`OPTIONAL`) attributes. this example would

 ©1996 *B-Core (UK) Ltd.*

produce a module parameterised over the set PERSON and the numbers max_base1 and max_base2, (the maximum number of objects to be stored in each base).

The optional SUPPORTS clause indicates that the generated system is to support the two implementations fifiI and mimiI, and as a consequence the generated '.ops' file will automatically have those operations not required by the two implementations commented out.

In general, system description may contain a GLOBAL clause and/or one or any number of BASE clauses. Each clause contains one or several declarations. The declarations may be of the following forms:

7.1.1 GLOBAL declarations

- The declaration `aa : STRING[nn]` generates the specification of a variable `aa : STRING` (where STRING is the set of character strings) and operations on it; nn is taken to be the maximum length of aa.
- The declaration `aa : FSTRING[nn]` generates the specification of a variable `aa : STRING` (where STRING is the set of character strings) and operations on it; aa is, however, considered to be a fixed-length string and nn is taken to be the fixed length of aa. Strings of length less than nn can be stored in aa.
- The declaration `aa: SEQ(CC)[nn]` generates the specification of a variable `aa : seq(CC)` and operations on it; nn is taken to be the maximum length of aa.
- The declaration `aa: SET(CC)[nn]` generates the specification of a variable `aa <: CC` and operations on it; nn is taken to be the maximum size of aa.

7.1.2 BASE declarations

Let bb be the name of a base.

- The declaration `aa: STRING[nn]` within a MANDATORY clause generates the specification of a variable `aa : bb --> STRING` and operations on it; nn is taken to be the average length of aa.
- The declaration `aa: STRING[nn]` within an OPTIONAL clause generates the specification of a variable `aa: bb +-> STRING` and operations on it; nn is taken to be the average length of aa.
- The declaration `aa: FSTRING[nn]` within a MANDATORY clause generates the specification of a variable `aa : bb --> STRING` and operations on it; aa is, however, considered to be a fixed-length string and nn is taken to be the fixed length of aa. Strings of length less than nn can be stored in aa.

 ©1996 *B-Core (UK) Ltd.*

- The declaration `aa: FSTRING[nn]` within an OPTIONAL clause is ILLEGAL.
- The declaration `aa: SEQ(CC)[nn]` within a MANDATORY clause generates the specification of a variable `aa: bb --> seq(CC)` and operations on it; `nn` is taken to be the average length of `aa`.
- The declaration `aa: SEQ(CC)[nn]` within an OPTIONAL clause generates the specification of a variable `aa: bb +-> seq(CC)` and operations on it; `nn` is taken to be the average length of `aa`.
- The declaration `aa: SET(CC)[nn]` within a MANDATORY clause generates the specification of a variable `aa: bb --> POW(CC)` and operations on it; `nn` is taken to be the average size of `aa`.
- The declaration `aa: SET(CC)[nn]` within an OPTIONAL clause generates the specification of a variable `aa: bb +-> POW(CC)` and operations on it; `nn` is taken to be the average size of `aa`.
- The declaration `aa: CC` within a MANDATORY clause generates the specification of a variable `aa: bb --> CC` and operations on it. `CC` will become a parameter to the generated machines.
- The declaration `aa: CC` within an OPTIONAL clause generates the specification of a variable `aa: bb +-> CC` and operations on it. `CC` will become a parameter to the generated machines.
- The declaration `aa: basename` within a MANDATORY clause generates the specification of a variable `aa: bb --> basename` and operations on it.
- The declaration `aa: basename` within an OPTIONAL clause generates the specification of a variable `aa: bb +-> basename` and operations on it.

For each base name 'basename' in the description a variable

```
basename <: basename_ABSOBJECT
```

is generated together with operations which changes the value of basename.

The generated machine is parameterised by:

1. all sets (upper case set identifiers) appearing in the SYSTEM description as 'types' (e.g. `CC` in `SEQ(CC)[nn]`)
2. a '`max_basename`' for each BASE name '`basename`' appearing in the SYSTEM description

 ©1996 *B-Core (UK) Ltd.*

The implementation is built on the SLIB library: a '`Vffnc`' machine is imported for the GLOBAL variables, and a separate '`fnc_obj`' is imported for each BASE; if the SYSTEM description contains `STRING`, `SET` or `SEQ`, a single '`str_obj`', '`set_obj`' or '`seq_obj`' respectively is also imported. Thus each (variable length) string, set or sequence will be stored in its appropriate 'obj' machine, with the tokens denoting these objects being stored in the '`Vffnc`' or '`fnc_obj`' machine; all other objects are stored directly in the '`Vffnc`' or '`fnc_obj`' machine

An `FSTRING` differs from a `STRING` in the manner in which it is implemented; the former is stored, in compressed form, inside the '`fnc_obj`' machine itself, (whereas the latter is of variable length and stored in the '`str_obj`' machine, and its token stored in the '`fnc_obj`' machine). Further, operations for direct copying and testing for equality are provided for `FSTRING`s having the same length.

The Base Generator tools also perform BASE language syntax checking; the MiniRemake option is provided if an error is found.

 ©1996 *B-Core (UK) Ltd.*

7.2 The Enumerator (enm)

The Enumerator is invoked from the Generators Environment and takes as input a source enumeration file with file extension '.enm', for example fifi.enm, and outputs a configured Abstract Machine source file to the development CFG directory, for example fifi.mch, together with related analysed, type and C program files to the ANL, TYP and CDE/C directories respectively.

Abstract Machines produced by the tool are standard library specifications for use in other constructs, therefore they are guaranteed to be syntactically correct, well-typed and internally consistent.

The purpose of the Enumerator is to provide basic teletype i/o operations for enumerated sets present in a development.

The general form of an enumeration description is given in a BNF-like syntax as follows: the syntax `<< exp >>` indicates zero or more occurrences of `exp` (repetition):

```
enumeration_description ::=  ENUMERATION
                               enumeration_name
                             SEES
                               sees_list
                             SETS
                               set_list
                             END

enumeration_name    ::=  Identifier

sees_list           ::=  Identifier << , Identifier >>

sets_list           ::=  Identifier << ; Identifier >>
```

The `SEES` clause identifies those specifications that are 'in scope' for the enumeration and the `SETS` clause identifies those enumerated sets for which operations are to be generated; each set should be defined in one of the machines in scope.

An example enumeration is:

```
ENUMERATION

  fifi

SEES
```

 ©1996 *B-Core (UK) Ltd.*

```
  mimi, gigi

SETS

  SET1; SET2

END
```

This defines a construct comprising two enumerated sets (`SET1` being an enumerated set of `mimi`, and `SET2` being an enumerated set of gigi). When submitted to the generator it produces a machine (and code) which may be employed in i/o operations. Three operations for each enumerated set are provided:

- PUT_SET1, PUT_SET2

  having a single input parameter, which value can range over the enumerated set; the operation is an output operation to standard terminal output and might be used in an interactive application to output the value of a particular variable which ranges over the enumerated set.

- GET_SET1, GET_SET2

  having a single output parameter, which value ranges over the enumerated set; the operation is an input operation which takes its input from standard keyboard input and can be used to input a particular value ranging over the enumerated set. The generated code is robust and features a small help facility.

- GET_PROMPT_SET1, GET_PROMPT_SET2

- having an input parameter, with precondition that it is a string, and output parameter, which value ranges over the enumerated set; the operation is an input operation which takes its input from standard keyboard input and can be used to prompt for a particular value ranging over the enumerated set. Again the generated code is robust and features a small help facility.

Note that the Enumerator is automatically employed by the non-Motif Interface generator where necessary.

 ©1996 *B-Core (UK) Ltd.*

7.3 The Interface Generator (itf)

The Interface tool is invoked from the Generators Environment and takes as input a source interface description file, with file extension '.itf'.

An option to introduce an 'Interface of Implemented Machine' is available under the Tool Bar Introduce.

The general form of an interface description is given in a BNF-like syntax as follows:
the syntax `exp1 | exp2` indicates `exp1` or `exp2` (choice),
`<< exp >>` indicates zero or more occurrences of `exp` (repetition) and
`[| exp |]` indicates optionality.

```
interface_description ::=   Interface!Generator
                              interface_name | interface_name(params)
                            [|
                               IMPORTS
                                 imp_construct << , imp_construct >>
                            |]
                            OPERATIONS
                              operation_list
                            END

interface_name        ::=  Identifier

imp_construct         ::=  Identifier ( param << , param >> )

param                 ::=  Identifier |  Bnumber

operation_list        ::=  Identifier << , Identifier >>
```

The interface name must be that of a MACHINE for which an appropriate analysed implementation exists (see below). The OPERATIONS clause identifies those operations which the user wants to appear in the interface (i.e. be 'in-scope' for testing) and should be valid operations of the referenced specification; an error will be reported otherwise.

If the MACHINE is parameterised, the parameters must be set in the Interface!Generator; an error will be reported otherwise.

Since the interface is the top level of a development, all machines which are SEEN but not IMPORTED at a lower level need to be imported at this interface level. If such a machine is parameterised, actual parameters must, of course, be supplied at import, and so the optional IMPORTS clause provides the point at which such parameters may be set.

The introduction of an 'Interface of Implemented Machine' option described above auto-

 ©1996 *B-Core (UK) Ltd.*

matically provides the correct description - in template form - of the interface, including the introduction of the IMPORTS clause where necessary. This is the recommended method of introducing an interface into a development; the only task for the user is to provide (where necessary) actual parameters, and the interface is the built completely automatically.

Two types of interface may be built, dictated by the setting of the Interface Option.

7.3.1 The non-Motif Interface

The Interface tool will generate an implementation which allows the user to test the named operations of the implementation developed by the user and corresponding to the named specification. That implementation should be built using the SLIB constructs and data types; other concrete data types will not be recognised by the Interface generator and will result in an error. Where the user's operations require input/have output, the interface implementation will produce appropriate prompts and call the relevant library procedures.

The Interface generator produces two specification/implementation pairs: for example if the Interface!Generator name if 'fifi', the following constructs are generated:

1. `fifiIo.mch` and `fifiIoI.imp`
   This specification/implementation pair concerns the i/o relating to the 'in-scope' operations; the specification comprises an enumerated set of operation names, with weakest specification operations to display the menu of operation names, and to select one of the names from the set. The implementation is achieved by importing using the 'basic_io' SLIB machine.

2. `fifiItf.mch` and `fifiItfI.imp`
   This pair has a single operation 'main' (specified as 'skip') and is implemented by importing (or seeing, depending on the way the user has built the implementation of the original specification) the specification for which the interface is being generated, the generated i/o machine, and the SLIB constructs 'basic_io', 'file_dump', 'token_io' and 'Bool_TYPE'; another generated construct, EnumItf is also imported, if required. The operation is implemented as an initialised loop, enabling the user to quit, display the menu or select an operation (calling the i/o machine), prompting for input (where appropriate), calling the selected operation of the users development, and displaying output (where appropriate).

The construct '`EnumItf.enm`' will also be produced if the user's specification contains an enumerated set. The ENUMERATION, if produced, enables i/o to be constructed for the user's enumerated sets (by using the Enumerator tool).

After each of these constructs have been generated, the appropriate jobs are placed on the job queue (enumeration, analysis, translation and linking), and, if the process is successful,

 ©1996 *B-Core (UK) Ltd.*

```
 0 Person Menu

 1 Add_person
 2 Get_person_details
 3 Remove_person
 4 Save_Person
 5 Restore_Person

 6 Quit

Person operation number? 1

 Input (STRING) Value for name_inp: jane
 Input (NAT) Value for age_inp: 28
 Input (SEX enumerated element) for sex_inp: female

 Value (BOOL) returned in rep: TRUE
 Token (PERSON/NAT) returned in new_person: 1

Person operation number? 2

 Input (PERSON/NAT) Token for person_inp: 1

 Value (BOOL) returned in rep: TRUE
 Value (STRING) returned in name_out: jane
 Value (NAT) returned in age_out: 28
 (SEX enumerated element) returned in sex_out: female

Person operation number?
```

Figure 17: Non-Motif Interface

it will result in executable code, accessed through the Translators Environment by calling the 'Run' tool with the generated interface implementation. Errors are dealt with in the usual way:

See figure 17.

7.3.2 The Motif Interface

The Interface tool will generate and compile Motif source code directly; the resulting code may be accessed in the Translators Environment by calling the 'Run' tool with the generated interface.

This option is much faster than the non-Motif version (since no intermediate Abstract Machines are generated, and there is consequently no need for enumeration, analysis, and translation), but a Motif development environment must be installed on the user's system for this option to function.

The resulting Motif interface pops up a window presenting a push-button menu of operation

Figure 18: Motif Interface Operations Menu

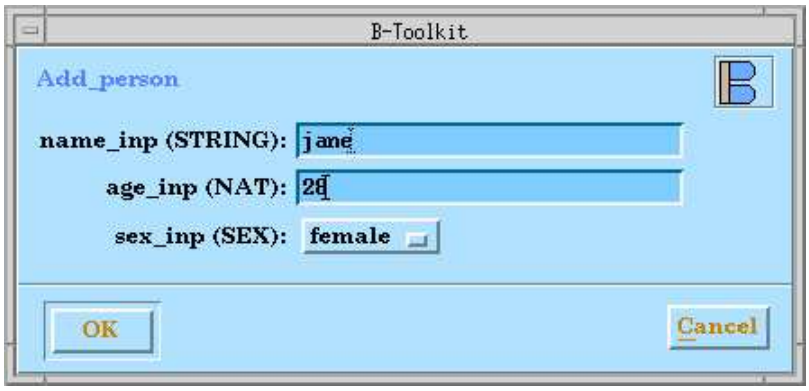


Figure 19: Motif Interface Input

names:

See figure 18.

and prompts for input as a form (i.e. operations with more than one input variable appear as a single form to fill in), with enumerations appearing as cascading menus:

See figure 19.

Output is displayed in the Display Area:

See figure 20.

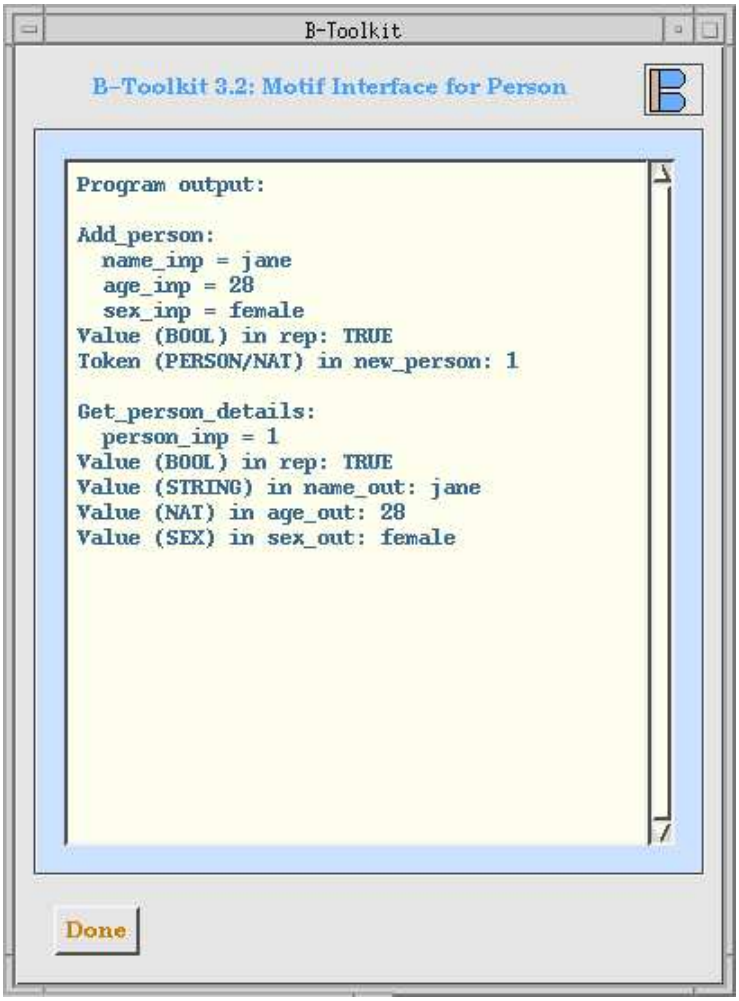


Figure 20: Motif Interface Display

 ©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

B-TOOLKIT USERS' MANUAL

SECTION 8

Code

©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

8 Code

8.1 The Translator (trl)

The Translator (trl) is invoked from the Translators Environment and is applicable to all implementations which are currently analysed, but not translated.

The environment variable BCC may be set to override the default setting for calling the C-compiler, which is 'cc'. For example:

```
setenv BCC "gcc -g"
```

would result in code produced by the gcc compiler with the debugging option set.

The setting of the Programming Language Option determines whether (currently) C or ANSI-C is produced as the intermediate language.

The Translator translates an IMPLEMENTATION into an executable code module; a prerequisite for translation is that the implementation and all subordinate implementations are analysed; the C Compiler is used to generate the executable binary for the machine on which the B-Toolkit is running. Subordinate implementations that are analysed but not yet translated are translated first.

If the specification 'spec.mch' is implemented by 'prog.imp', then translation of the latter will produce the files 'spec.g','spec.h', 'spec.c' and 'spec.o' all written to the CDE/C directory.'spec.o' is the binary linkable code module, 'spec.h' contains header information for an IMPLEMENTATION which IMPORTS or SEES 'spec.mch', and 'spec.g' contains module parameterisation information to be used at link-time when the values of module parameters are known.

Note that, an IMPLEMENTATION has no variables of its own which are global to all operations (e.g. the VARIABLES clause is not allowed). All static data (information global to the module) or persistent data (information which survives from one invocation of the module to the next) must be held within other machines which the implementation IMPORTS. In the B-Toolkit environment there is a selection of such pre-defined machines.

The pre-defined machines are held in the System Library and can be introduced into a development by using the 'Introduce construct from SLIB' facility.

8.1.1 The 'main' Implementation:

The machine/implementation which contains the single operation 'main' will be the main module of the development. This module will be the top node when all executable object

 ©1996 *B-Core (UK) Ltd.*

modules are linked, and the main operation will be the entry when the linked code is invoked.

8.1.2 The Translation Process

The Translator translates the following clauses of an IMPLEMENTATION:

- The MACHINE header clause:
  The Translator provides for an implementation of the formal machine parameters. Linking information is generated from this clause.

- The IMPORTS clause:
  The Translator provides for an implementation which makes instances of the imported MACHINEs known to this implementation; an imported MACHINE must be a library (SLIB or TLIB) machine, a generated machine (with code module, for example generated through the Enumerator) or a MACHINE which already has been provided with an IMPLEMENTATION within the development. Linking information is generated from this clause.

- The SEES clause:
  The Translator provides for an implementation which makes seen MACHINEs known to the implementation. A seen MACHINE must be a committed construct which is imported elsewhere, in which case its state is shared between the machine which imports it and the machine which sees it.

- The CONSTANTS and PROPERTIES clauses:
  All constants must be given an exact value in the PROPERTIES clause. The constants and its values are known to all implementation which SEES or IMPORTS the MACHINE for which we are providing an implementation. Header information is generated from this clause.

- The SETS and PROPERTIES clauses:
  The Translator provides for an implementation of enumerated sets and deferred sets. The deferred sets must be given an exact value within the latter clause; however the value is normally a deferred set from an imported machine. Header information is generated from this clause.

- The INITIALISATION clause:
  The Translator provides for a C implementation of the initialisation which is executed at run-time on invocation. Note that the INITIALISATION clauses of all imported MACHINES are automatically invoked before any statements present in this clause.

 ©1996 *B-Core (UK) Ltd.*

- The OPERATIONS clause:
  Each of the operations is translated into code. Note that for imported SLIB MACHINES, operations are translated into parameterised macros and others into C functions or procedures.

8.1.3 Note:

- The 'file_dump' machine must always be imported into the development when the development uses a state machine from the System Library, even if the dump facilities SAVE and RESTORE from the state machine are not actually used.

©1996 *B-Core (UK) Ltd.*

8.2 The Linker (lnk)

The Linker is invoked from the Translators Environment and C-compiles a suitable implementation into object code.

A prerequisite for linking is that the implementation has been translated.

The environment variable BCC may be set to override the default setting for calling the C-compiler, which is 'cc'. For example:

```
setenv BCC "gcc -g"
```

would result in code produced by the gcc compiler with the debugging option set.

If the specification 'fifi.mch' is implemented by 'mimi.imp', then linking of the latter will produce the executable file 'fifi', written to the CDE/C directory.

The linker must be invoked against the implementation which defines the `main`, procedure, otherwise an error occurs. An attempt to link an implementation having parameters (which will, of course, have been defined in its top-level abstraction) will result in an error, as will attempting to link an implementation in which machines SEEN at a lower level have not all been imported.

See also the Translator and notes on Structuring Developments.

 ©1996 *B-Core (UK) Ltd.*

8.3 Run (run)

The Run command is invoked from the Translators Environment and is applicable to implementations which are currently linked, or machines for which Motif interfaces which are currently generated.

For the former, the tool provides a window in which execution of the code takes place; the latter automatically provides an X/Motif environment.

 ©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

B-TOOLKIT USERS' MANUAL

SECTION 9

Documents

©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

9 Documents

9.1 The Document Mark-Up Tool (dmu)

DocumentMarkUp (dmu) is invoked from the Documents Environment and is applicable to all constructs that have not been marked up (with the exception of '.ops' constructs).

The tool currently produces LaTeX source code (a source file named constructname.tex written in the TEX directory) and the system command 'latex' is issued. This command is intended to invoke latex2e, and the user should ensure that the PATH environment variable is appropriately set.

The formatting of AMN constructs is automatic; the user may, however, dictate the style by the positioning of newlines/annotations in the document.

Note that the following comments do not apply to proof files; see Top Bar Help on Proof-Printer concerning the editing of proof files.

9.1.1 Comments and Annotations

The AMN source document may contain normal C-like comments:

```
/* comment */
```

which are removed before processing.

However a special kind of comment - an annotation - is also permitted:

```
/*" annotation "*/
```

Annotations are not discarded, but processed, and form an integral part of the end document, and so provide a means of complementing the formal mathematics of AMN documents with informal comment.

Annotations comprise LaTeX source code, and thus any LaTeX commands may be used inside an annotation, for example:

```
/*" {\bf A bold statement}\\
... some annotation ... "*/
```

A single newline after the begin-annotation or preceding the end-annotation is not significant; thus the following annotations will appear the same in the printed document:

 ©1996 *B-Core (UK) Ltd.*

```
/*" annotation "*/

/*"
   annotation
"*/
```

More that one newline in either position will result in more space appearing at that point in the document.

Annotations will normally be separated from the mathematical content in a document by horizontal lines; horizontal lines will not, however, appear around an empty annotation - one consisting entirely of white space (which may therefore be used to print vertical space) , or around an newpage annotation - one consisting of the command '`\newpage`' and optional white space (which will, of course, cause a newpage to be produced).

9.1.2 Positioning of Comments

- Comments may appear anywhere in a document
- Comments may not be nested
- Comments may not appear inside annotations

9.1.3 Positioning of Newlines

- Newlines may appear anywhere in a document, however only those detailed below are echoed in the LaTeX document.
- Optional newlines may appear after the following construct keywords:

MACHINE	REFINEMENT	IMPLEMENTATION	ENUMERATION
SYSTEM	BASE	CONSTRAINTS	
USES	SEES	REFINES	INCLUDES
IMPORTS	PROMOTES	EXTENDS	SETS
CONSTANTS	PROPERTIES	VARIABLES	INVARIANT
ASSERTIONS	DEFINITIONS	INITIALISATION	OPERATIONS

  Each of the above keywords will always be preceded by a newline in the LaTeX document
- Optional newlines may appear before or after the following operation keywords:

 ©1996 *B-Core (UK) Ltd.*

```
BEGIN   PRE   THEN    WHEN     IF        ELSE
ELSIF   ANY   WHERE   SELECT   CASE      BE
EITHER  OR    LET     CHOICE   IN        WHILE
DO      VAR   OF      VARIANT  INVARIANT
```

- The keyword END will always be preceded by a newline in the LaTeX document except when no newline appears in the following constructs:

```
BEGIN ... END
PRE ... THEN ... END
IF ... THEN ... END
IF ... THEN ... ELSE ... END
ANY ... WHERE ... THEN ... END
```

- Optional newlines may appear before or after any of the following symbols:

```
&   ;   ,   ||   =   =>   <=>   or   :=
```

- Breaking long lines may be achieved by the use of brackets; an open bracket immediately followed by a newline will cause a linebreak with all following lines being indented, and a closing bracket immediately preceded by a newline will stop the indentation. So a predicate written:

```
xxx => (
yyy
 .
 .
 .
zzz
)
```

will be marked up with the consequent list indented as follows:

```
xxx => (
  yyy
   .
   .
   .
  zzz)
```

Note that if the brackets are unnecessary they will be discarded, although the linebreaking and indentation will still occur.

- Multiple newlines are treated as a single newline

 ©1996 *B-Core (UK) Ltd.*

9.1.4 Rules concerning Annotations

- No white space should appear between annotation-defining characters `/ * "`. For example:

  ```
  /* " annotation "*/
  ```

  would be treated by the tool as a comment, and therefore discarded.

- The start-annotation symbol `/*"` must appear at the beginning of a line
- The end-annotation symbol `"*/` must appear at the end of a line
- Annotations may not be nested
- Annotations may not appear inside a comment
- Annotations may appear in the following positions:
  - before or after each of the construct keywords listed above
  - before or after after each of the operation keywords:

    ```
    THEN   WHEN  ELSE   ELSIF   WHERE  BE  OR  IN  OF
    ```

  - after each of the operation keywords:

    ```
    BEGIN   PRE     IF     ANY  SELECT  CASE     EITHER
    LET     CHOICE  WHILE  DO   VAR     VARIANT  INVARIANT
    ```

  - before or after after each of the symbols:

    ```
    &   ;   ,   ||   :=
    ```

  - If no annotation appears immediately preceding an operation, the operation name will be automatically printed in bold preceding the operation

9.1.5 If LaTeX complains

Should your file fail to LaTeX correctly (for example if an annotation contains bad syntax) you should quit from LaTeX by typing 'x' in the LaTeX window, edit the file, commit the edits and re-submit it for mark-up.

Note that the double quote character " inside an annotation causes the message 'Bad annotation position' to be displayed (this is because annotations are represented internally as strings). Single quotes ‘ and ’ may be used; if double quotes are required, ‘‘ and ’’ should be used.

 ©1996 *B-Core (UK) Ltd.*

9.1.6 The Document Options

- If the 'Labels' option is set, the LaTeX commands `\ref` and `\pageref` may be used inside annotations to reference the occurrence of construct, variable, set, constant and operation names (for MACHINES, REFINEMENTS and IMPLEMENTATIONS)
- If the 'Clause cross-references' option is set, a cross-reference of external construct, variable, set, constant and operation names will automatically be produced after the PROPERTIES, INVARIANT ASSERTIONS, INITIALISATION clauses (if present) and following each operation (for MACHINES, REFINEMENTS and IMPLEMENTATIONS)
- If the 'Construct cross-references' option is set, a cross-reference of external construct, variable, set, constant and operation names will automatically be produced at the end of each MACHINE, REFINEMENT and IMPLEMENTATION
- If the 'Index' option is set, an index will be produced containing references to all construct, variable, set, constant and operation names at the end of each DOCUMENT

See also Options.

 ©1996 *B-Core (UK) Ltd.*

9.2 Documents

A DOCUMENT is an AMN construct, having a `.doc` file extension, which enables the preparation of a single composite document incorporating other AMN construct, (possibly other documents) which may be submitted to the Document Mark Up tool.

A document is displayed only in the Documents Environment.

An option to introduce a 'Document of configured constructs' is available under Introduce on the the Tool Bar.

The general form of a DOCUMENT is given in a BNF-like syntax as follows:
the syntax `exp1 | exp2` indicates `exp1` or `exp2` (choice), and
`<< exp >>` indicates zero or more occurrences of `exp` (repetition).

```
document_description  ::=  DOCUMENT
                             document_name
                           CONTENTS
                             includes_list
                           END

document_name         ::=  Identifier

includes_list         ::=  includes(construct)
                               << , includes(construct) >>

construct             ::=  Identifier.extension |
                           Identifier.extension(op_list)

op_list               ::=  Identifier << , Identifier >>

extension             ::=  mch | ref | imp | bse |
                           int | enm | doc | mch.number.prf |
                           ref.number.prf | imp.number.prf
```

Annotations may appear anywhere inside the CONTENTS clause.

The optional op_list argument dictates which operations are to be printed. If it is not present, all operations are printed, otherwise only those appearing in the list are printed. If none of the operations are to be printed, this may be achieved by including an op_list comprising a single name that is not one of the operations of that construct.

When a DOCUMENT is submitted for mark up, all included constructs that have already been marked up are copied into the marked up document; thus the characteristics of that already marked up construct remain (for instance the presence or otherwise of cross

referencing - see Options). The current characteristics may be obtained by first resetting all documents (available under Utilities.

Each included document that has not been marked up is first marked up, if possible. For example, if 'fifi.mch' is included, but is not yet analysed (and thus not yet ready for mark up) a warning is issued, and fifi.mch would not be included in the marked up DOCUMENT.

An index can be automatically produced at the end of a DOCUMENT by setting the Document Index flag.

©1996 *B-Core (UK) Ltd.*

9.3 Show (shw)

This option is invoked from the Documents Environment and is applicable for all constructs that are currently marked-up.

It enables a '.dvi' file to be viewed on-screen through the setting of the environment variable BSDVI; for example a suitable setting might be:

```
setenv BSDVI "xdvi -s 4"
```

Of course, it may be necessary to set further environment variables as a result, for example XDVIFONTS.

©1996 *B-Core (UK) Ltd.*

9.4 Print (prt)

This option is invoked from the Documents Environment and is applicable for all constructs that are currently marked-up.

It enables a '.dvi' file to be sent to a printer through the setting of the environment variable BPDVI; for example a suitable setting might be:

```
setenv BPDVI /usr/local/bin/printdvi
```

where the contents of the file 'printdvi' might be:

```
#! /bin/sh
dvi2ps $1 > $1.ps
lpr -Phplaser $1.ps
rm -f *.ps *.aux *.dvi *.log *.toc
lpq -Phplaser
```


©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

B-TOOLKIT USERS' MANUAL

SECTION 10

Libraries

©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

10 Libraries

10.1 The System Library (SLIB)

The B-Toolkit is supplied with a set of reusable machines which encapsulate basic state and data-structures, and provide basic abstract types.

These library machines are described here. The text of the machines is presented separately.

10.1.1 The Role of the System Library

Static and persistent data of an implementation must be held by an imported machine. Ultimately, the bottom level implementation must import a Library machine if static or persistent data is required (i.e. if you are implementing a state-machine).

A library machine is used simply by importing or seeing the machine in the construct you are writing. When a predefined machine is imported into an implementation, actual parameters must be supplied. However when seeing an existing machine no parameters should be supplied.

Each library machine is formally specified, and the readable formal specification is available in the development environment after the machine has been committed.

The manipulation of an object can be performed outside the state-machine to which the object belongs, using the TYPE machines of the Library.

Simple i/o can be performed using the i/o machines.

Persistence of data can be achieved using the file i/o machines.

10.1.2 Naming of Library Machines and their Operations

The naming library machines and their operations follows a uniform scheme.

Library machines with state have to be renamed when they are imported into a development. These machines and their operations, therefore, all have names starting with `Rename`, which is automatically replaced by a new prefix supplied in the importing machine at the time that the library machine is committed from the library. Note that the rename prefix may not contain the underscore '_' character.

The TYPE machines in the library have no state, and need not be renamed. They can be cited in a SEES clause wherever they are required.

Operations in library machines are named according to a uniform pattern described in the

 ©1996 *B-Core (UK) Ltd.*

text below.

10.1.3 Library Machines

Library Machines are organised into five groups as follows.

- Multiple Objects Machines
- Programming Concepts Machines
- Mathematical Concepts Machines
- Input Output Machines
- Type Machines

10.1.4 Multiple Objects Machines

A Multiple Object Machine is a repository for structured objects. In this sub-library we have:

- Rename_set_obj, a machine for collections of Sets
- Rename_seq_obj, a machine for collections of Sequences.
- Rename_str_obj, a machine for collections of Strings of Bytes.
- Rename_fnc_obj, a machine for collections of Functions over Natural Numbers.
- Rename_ffnc_obj, a machine for collections of Functions over Natural Numbers with multiple field operations.

In most applications of B to software development, the Multiple Objects Machines are used for representing the main part of the information which an application maintains. Composite object structures which are suitable for representing complex information structures can also be constructed.

For example, by using combinations of the Multiple Objects Machines we can construct a suitable representation for files of records with different field types. Types can be simple types (SCALAR, BOOL, BITS) or more complex (Sets of Strings, Sequences of Sets of Strings, Sets of Sequences of BITS etc). Such constructions gives the formal B design language the equivalent power of a modern high level programming notation.

When importing a Multiple Object Machine it must be instantiated with respect to the size of store which will be pre-allocated for storing its objects, and with respect to the type of elements which actually will populate the structured objects. A multiple object

 ©1996 *B-Core (UK) Ltd.*

machine can store a number of objects of the required type (e.g. sequences of BOOL, sets of SCALARS, sets of SEQOBJ).

Composite object structures can be constructed. For example a 'Sequences of Sets of SCALARS' is constructed by using a Set Object machine instantiated with SCALAR in combination with a Sequence Object machine instantiated with SETOBJ.

When providing a design using a machine as a repository then elements of *different* types can be stored within the *same* machine; e.g. a sequence object machine instantiated with 'BOOL or SCALAR' (written `BOOL \/ SCALAR`) can store Boolean values as well as scalar values (positive integers) within the same sequence.

All Multiple Objects Machines offer the following facilities:

- Initialisation (INI): This operation resets the repository to its initial empty state.
- Creation (CRE) and killing (KIL) of objects: On creation of a new object a unique token will be allocated to denote the created object (e.g SEQOBJ are the tokens allocated for Sequence Objects). If the creation has been successful this token will be returned on creation, and required as input for any operation concerning this object (e.g. killing).
- Capacity Enquiries (MEM_FUL, OBJ_FUL): All these operations can be used to determine whether the capacity of the repository has been reached.
- Token Enquiries (ANY, XST): ANY returns a token of proper type (e.g. the Sequence Machine's ANY operation returns a SEQOBJ token). The XST operation can be used to determine whether a token denotes an object which actually exists.
- Persistent storage (SAV, RST): The save operation (SAV) saves the current state of the object machine onto permanent storage, while the restore operation (RST) restores the object machine to a previously saved state.

  Note that when saving several structures on the same file then the restores *must* be performed in the same order as the saves were performed; different dump files can be used by using **file_io** operations.
- Browsing facilities (FIRST, NEXT): A browse through all stored objects is initiated by using the FIRST operation, which will return an object (if any) and a cursor for continuing the browse by using NEXT. When the cursor reaches '1' the last object has been retrieved.

10.1.5 Machine Rename_set_obj

The general multiple object operations, which are described above, are provided:

 ©1996 *B-Core (UK) Ltd.*

- Initialisation (INI).
- Creation (CRE) and killing (KIL) of sets; on creation the created set will be empty.
- Capacity Enquiries (MEM_FUL, OBJ_FUL)
- Set token Enquiries (ANY, XST).
- Persistent Set storage facilities (SAV, RST).
- Sets Browsing facilities (FIRST, NEXT).

In addition, this machine offers special operations for set manipulation and enquiring about the stored sets:

- Operations for changing existing sets: (CPY, CLR) CPY copies one set to another set; CLR removes all elements from a set.
- Operations for enlarging sets: (ENT, UNION) ENT enters a new element into a set; UNION extends a set by all the members of another set.
- Operations for reducing sets: (RMV, DIFF, INTER) RMV removes an element from a set; DIFF removes all elements which are also present in another set; INTER keeps only the elements which are present in another set.
- Enquiring about size of sets: (EMP, CRD) EMP returns TRUE if the set interrogated is empty; CRD gives the cardinality of a set.
- Facilities for inspecting a particular set: (XST_ORD, VAL) All elements in all sets can be accessed by using their ordinal number. XST_ORD will indicate whether a particular number is a valid ordinal number. VAL will return the value of an element in the set, given its ordinal number.
- Operations for determining set properties: (MBR, INCL) MBR will indicate whether a particular element is a member of a set; INCL will indicate whether one set is included in another.

10.1.6 Machine Rename_seq_obj

The general multiple object operations, which are described above, are provided:

- Initialisation (INI).
- Creation (CRE) and killing (KIL) of sequences. On creation the created sequence will be empty.

 ©1996 *B-Core (UK) Ltd.*

- Capacity Enquiries (MEM_FUL, OBJ_FUL)
- Sequence token Enquiries (ANY, XST).
- Persistent Set storage facilities (SAV, RST).
- Sequence Browsing facilities (FIRST, NEXT).

In addition, this machine offers special operations for manipulation of sequences, and enquiring about stored sequences:

- Operations for changing existing sequences: (CPY, CLR, OVR, STO) CPY copies one sequence to another sequence; CLR resets a sequence to the empty sequence; OVR over-writes a sequence with another sequence; STO over-writes a particular location in the sequence.
- Operations for re-ordering an existing sequence: (SWA, REV) SWA swaps two elements in a sequence; REV reverses an entire sequence.
- Operations for enlarging sequences: (PSH, APP) PSH pushes an element onto the end of a sequence; APP extends a sequence with another sequence.
- Operations for reducing sequences: (KEP, CUT) KEP keeps an initial part of a sequence; CUT cuts an initial part of a sequence away.
- Enquiring about a sequence: (EMP, LEN) EMP returns TRUE if a sequence is empty; LEN gives the length of a sequence.
- Facilities for inspecting a particular sequence: (POP, XST_IDX, VAL) POP returns the most recently pushed element; an element in a sequence can be accessed by using its index in its sequence; XST_IDX will indicate whether a particular number is a valid index; VAL will return the value of an element in the sequence given its index.
- Operations for determining sequence properties: (MBR, EQL) MBR will indicate whether a particular element is a member of a sequence. EQL will indicate whether two sequences are identical.

10.1.7 Machine Rename_str_obj

The String Machine is a repository for sequences of bytes (seq(0..255)). It is often used for storing text strings. The facilities for strings offered in the library and the the B translators enable strings to be treated as simple objects, which can be transferred from one machine to another using operation which will transfers the entire structured string object. (See also the Fixed Function Machine)

The general multiple object operations, which are described above, are provided:

- Initialisation (INI).
- Creation (CRE) and killing (KIL) of strings; on creation the created string will be empty.
- Capacity Enquiries (MEM_FUL, OBJ_FUL)
- String token Enquiries (ANY, XST).
- Persistent String storage facilities (SAV, RST).
- String Browsing facilities (FIRST, NEXT).

In addition, this machine offers special operations for manipulation of strings, and enquiring about stored strings:

- An additional operation for creating a particular string: (NEW) NEW will create a new string with a particular contents, which either literally occurs in the design text, or, is the contents of a string which is a result of an input operation, or, is the contents of a string extracted elsewhere.
- An additional operation for extracting string contents: (XTR)
- Operations for changing existing strings: (CPY, CLR, OVR, STO) CPY copies one string to another string; CLR sets a string to the empty string; OVR over-writes a string with another string; STO over-writes a particular location in the string.
- Operations for re-ordering an existing string: (SWA, REV) SWA swaps two bytes in a string; REV reverses an entire string.
- Operations for enlarging strings: (PSH, APP) PSH pushes a byte onto the end of a string; APP extends a string with another string.
- Operations for reducing strings: (KEP, CUT) KEP keeps an initial part of a string; CUT cuts an initial part of a string away.
- Enquiring about size of strings: (EMP, LEN) EMP returns TRUE if a string is empty; LEN gives the length of a string.
- Facilities for inspecting a particular strings: (POP, XST_IDX, VAL) POP returns the most recently pushed byte; a byte in a string can be accessed by using its index in its string. XST_IDX will indicate whether a particular number is a valid index. VAL will return the value of a byte in the string, given its index.
- Operations for determining string properties: (SMR, EQL, EQL_LIT) SMR will indicate whether one string is lexicographically smaller than another string. EQL will indicate whether two store strings are identical. EQL_LIT will indicate whether a stored string is identical to another literal string (not stored in by this machine).

 ©1996 *B-Core (UK) Ltd.*

10.1.8 Machine Rename_fnc_obj

This machine stores partial Functions over an initial interval of Natural Numbers. The interval (its fields) will be determined when the machine is instantiated.

The general multiple object operations, which are described above, are provided:

- Initialisation (INI).
- Creation (CRE) and killing (KIL) of objects. On creation the created object will be empty.
- Capacity Enquiries (FUL)
- Function token Enquiries (ANY, XST).
- Persistent String storage facilities (SAV, RST).
- Object Browsing facilities (FIRST, NEXT).

In addition, this machine offers special operations for manipulation of functions, and enquiring about stored functions:

- Enquiry operations: (TST_FLD, DEF_FNC) TST_FLD will indicate whether a given number input is a field of the function. DEF_FNC will indicate whether a function has been given a value for a particular field number.
- Operations for changing a function: (STO, RMV) STO assigns a value to a particular field of a particular function; RMV removes a field from a function - the function will then be undefined for this field.

10.1.9 Machine Rename_ffnc_obj

This machine stores partial Functions over an initial interval of Natural Numbers. The interval (its fields) will be determined when the machine is instantiated. The general multiple object operations, which are described above, are provided:

- Initialisation (INI).
- Creation (CRE) and killing (KIL) of objects. The created function will be empty.
- Capacity Enquiries (FUL)
- Function token Enquiries (ANY, XST).

 ©1996 *B-Core (UK) Ltd.*

- Persistent Object storage facilities (SAV, RST).
- Object Browsing facilities (FIRST, NEXT).

This machine also offers the operations for manipulation of functions, and enquiring about stored functions as are available for the standard Function Object Machine:

- Enquiry operations: (TST_FLD, DEF_FNC) TST_FLD will indicate whether a given number is a field of a function. DEF_FNC will indicate whether a function has been given a value for a particular field number.
- Operations for changing a function: (STO, RMV) STO assigns a value to a particular field of a particular function; RMV removes a field from a function - the function will then be undefined for this field.

In addition, special operations for multiple field manipulation are provided:

- Moving several field values from one function to another: (MOV_FFNC) MOV_FFNC will over-write the field values of one function with field values from another function.
- Overwriting several field values: (OVR_FFNC) OVR_FFNC will overwrite several fields in a function with the packed version of a sequence of bytes (String).
- Extracting multiple field values: (XTR_FFNC) XTR_FFNC will extract several field values - converted into a sequence of bytes stored at a word boundary.
- Testing multiple fields: (EQL_FFNC) This tests whether the contents of sequence of bytes is equal to the sequence of bytes stored in multiple fields of a function.

10.1.10 Programming Concepts Machines

These machine offer an extension of the basic data structures of **variable** and **array** which are available in most languages. We have:

- Rename_Vvar, a Variable of any type.
- Rename_Nvar, a Natural Number Variable.
- Rename_Varr, an Array of any type.
- Rename_Narr, an Array of Natural Numbers.

 ©1996 *B-Core (UK) Ltd.*

These machines are commonly used to record transitory information within an application, or used as 'anchors' for information stored in the Multiple Object Machines.

Variables can be instantiated to any type, e.g a variable of type BOOL can hold a boolean value, and a variable of type SEQOBJ (an anchor) can hold an token denoting a sequence within the Sequence Object Machine.

When providing a design using a machine as a repository, then elements of *different* types can be stored within the *same* machine. For example instantiating a machine with BOOL \/ SCALAR will enable a machine to have both boolean values as well as scalar values within a single structure.

All Programming Concepts Machines offer the following facilities:

- Getting a value (VAL) VAL will return the value of a variable, or in the case of an array the value of a particular location.
- Storing a value (STO) STO will assign a new value to a variable or in the case of an array to a particular location within the array.
- Testing for values (EQL, NEQ) EQL and NEQ will test the contents of a variable (or a particular location within an array) against another value.
- Persistent storage facilities (SAV, RES) SAV will save the current value of an object onto permanent storage; RES will restore the value of an object to a value previously saved.

Note that when saving several structures on the same file then the restores *must* be performed in the same order as the saves were performed; different dump files can be used by using **file_io** operations.

Also note that these primitive operations appear as 'procedure calls' within the algorithms of a B implementation; however in the generated implementation (the C-Code) all the simple operations from the Programming Concepts Machines are 'in-lined' (no procedure calls are involved).

10.1.11 Machine Rename_Vvar

This Variable machine offers all the facilities described above, and can be instantiated to any type (SCALAR, BOOL, BITS, SETOBJ, etc.) We have:

- Getting a value (VAL)
- Storing a value (STO)

 ©1996 *B-Core (UK) Ltd.*

- Testing for values (EQL, NEQ)
- Persistent storage facilities (SAV, RES)

10.1.12 Machine Rename_Nvar

This Variable machine, can hold only Numbers, and offers all the facilities available for a Programming Concepts Machine:

- Getting a value (VAL)
- Storing a value (STO)
- Testing for values (EQL, NEQ)
- Persistent storage facilities (SAV, RES)

In addition the following operations are provided:

- Arithmetic operations (INC, DEC, ADD, MUL, SUB, DIV, MOD) A variable can be modified by: INCrementing it, DECrementing it, ADDing to it, MULtiplying it by a number, SUBtracting a value from it, DIViding it with a number, or replacing it with a MODulo value.
- Ordering Queries (GTR, GEQ, SMR, LEQ) A variable can be tested to determine whether it is GreaTeR then a particular number, Greater or EQual to a number, SMalleR than a number, or, Less than or EQual to a number.
- Conditional retrieval (MIN, MAX) MIN will retrieve the smaller of two values, while MAX will retrieve the larger of two values.

10.1.13 Machine Rename_Varr

This Array machine can be used to store several values of any type. The type is determined by instantiation.

This machine offers all the facilities available for a Programming Concepts Machine:

- Getting a value (VAL) Getting the value under a particular index of the array.
- Storing a value (STO) Storing a value in a particular location of the array.
- Testing for values (EQL, NEQ) Testing the value stored under a particular index.

 ©1996 *B-Core (UK) Ltd.*

- Persistent storage facilities (SAV, RES)

In addition the following operations are provided:

- Index testing (TST_IDX) TST_IDX will indicate whether a particular index is a valid index.
- Search operations (SCH_LO_EQL, SCH_LO_NEQ, SCH_HI_EQL, SCH_HI_NEQ) These operation will return the lowest (or highest) array index which is equal (or not equal) to a particular given value.
- Array manipulations (REV, RHT, LFT, SWP) REV will reverse the array; RHT and LFT will shift the values in an array either to the right (to higher indices) or left (to lower indices); SWP will swap two values in an array.

10.1.14 Machine Rename_Narr

This Array machine can be used to store several Numbers.

This machine offers all the facilities available for a Programming Concepts Machine:

- Getting a value (VAL) Getting the value under a particular index of the array.
- Storing a value (STO) Storing a value in a particular location of the array.
- Testing for values (EQL, NEQ) Testing the value stored under a particular index.
- Persistent storage facilities (SAV, RES)

In addition the following operations are provided:

- Arithmetic operations (ADD, MUL, SUB, DIV, MOD) An array location can be modified by: INCrementing it, DECrementing it, ADDing to it, MULtiplying it by a number, SUBtracting a value from it, DIViding it with a number, or replacing it with a MODulo value.
- Ordering Queries (GTR, GEQ, SMR, LEQ) An array location can be tested to determine whether it is GreaTeR than a particular number, Greater or EQual to a number, SMalleR than a number or Less than or EQual to a number.
- Conditional retrieval (MIN, MAX) MIN will retrieve an array index, for which the array holds a minimum value, while MAX will retrieve an an array index, for which the array holds a maximum value.

 ©1996 *B-Core (UK) Ltd.*

In addition the following operations are provided:

- Index testing (TST_IDX) TST_IDX will indicate whether a particular index is a valid index.
- Search operations (SCH_LO_EQL, SCH_LO_NEQ, SCH_HI_EQL, SCH_HI_NEQ) These operation will return the lowest (or highest) array index which is equal (or not equal) to a particular given value.
- Additional Search operations (SCH_LO_GEQ, SCH_LO_GTR, SCH_LO_LEQ, SCH_LO_SMR, SCH_HI_GEQ, SCH_HI_GTR, SCH_HI_LEQ, SCH_HI_SMR) These operation will return the lowest (or highest) array index which is relates in a particular way (greater or equal, greater, less or equal, or, smaller) to a given value.
- Array manipulations (REV, RHT, LFT, SWP) REW will reverse the array; RHT and LFT will shift the values in an array either to the right (to higher indices) or left (to lower indices); SWP will swap two values in an array.
- Additional Array manipulations (SRT_ASC, SRT_DSC) Sort the array to contain either ascending values or descending values.

10.1.15 Mathematical Concepts Machines

The Mathematical Concepts Machines allow the programmer to design using higher level concepts, which are likely to match more closely the concepts used in a set theoretical specification, hence making the design step more reliable. We have:

- Rename_set, a Set machine.
- Rename_Vseq, a Sequence machine for a sequence of any type.
- Rename_Nseq, a Sequence machine for a sequence of Numbers.
- Rename_Vfnc, a Function machine of a particular range type.
- Rename_Nfnc, a Function machine for a Natural Number function.
- Rename_Vffnc, an Extended Function machine.

The Mathematical Concepts Machines are either used for storing transitory information or for 'anchors' to information stored within a Multiple Object Machine. When importing Mathematical Concepts Machine its type and its capacity must be given by instantiation. When providing a design using a machine as a repository then elements of *different* types can be stored within the *same* machine. For example instantiating a machine with

 ©1996 *B-Core (UK) Ltd.*

`BOOL \/ SCALAR` will enable a machine to have both boolean values as well as scalar values within a single structure. A Mathematical Concepts Machine offers the following common facilities:

- Getting a value (VAL) The VAL operation returns a particular value within the structure.
- Persistent Storage facilities (SAV, RES) For saving onto and restoring from permanent storage. **Note**When saving several structures on the same file then the 'restores' *must* be performed in the same order as the 'saves' were performed; different dump files can be used by using **file_io** operations.

10.1.16 Machine Rename_set

A set machine provides access to a single set structure. It offers the common operation for Mathematical Concepts Machine:

- Getting a value (VAL) with a particular ordinal number.
- Persistent Storage facilities (SAV, RST)

In addition it offers:

- Familiar Set Operations: (ENT, RMV, CLR) ENT enters a new element into the set; RMV removes an element from the set; CLR clears the set
- Familiar Set Enquiries: (MBR, EMP, CRD) MBR is used for testing set membership; EMP tests whether a set is empty; CRD gives the cardinality of a set.
- Additional Enquiries (FUL, XST_IDX) FUL indicates whether a set is full; XST_IDX indicates whether a particular number is a valid ordinal.

10.1.17 Machine Rename_Vseq

This sequence machine provides access to a single sequence. The type of the elements of the sequence can be determined through instantiation. It offers the common operations for Mathematical Concepts Machine:

- Getting a value (VAL) with a particular ordinal number.
- Persistent Storage facilities (SAV, RST)

 ©1996 *B-Core (UK) Ltd.*

In addition it offers:

- Familiar Sequence Operations: (CLR, PSH, POP, FST, LST, TAL, ST0) CLR clears the sequence; PSH pushes an element onto the sequence; POP gets the element most recently pushed; FST gets the first element; LST gets the last element; TAL modifies the sequence by removing the most recently pushed element; STO stores an element into the sequence.
- Some Sequence Enquiries: (EMP, LEN) EMP test whether the sequence is empty; LEN gives the length of the sequence.
- Some Sequence Tests: (NEQ, NEQ) EQL tests whether a particular value is at a particular index; NEQ tests whether a particular value is not at a particular index.
- Additional Enquiries: (FUL, XST_IDX) FUL indicates whether a sequence is full; XST_IDX indicates whether a particular number is a valid index.
- Searching Operations: (SCH_LO_EQL, SCH_LO_NEQ, SCH_HI_EQL, SCH_HI_NEQ) The searching operation search for the lowest (or highest) index in the sequence which is equal (or not equal) to a particular value.
- Sequence manipulation operations: (KEP, CUT, SWP, REV, RHT, LFT) KEP keeps the beginning of a sequence; CUT cuts the beginning of a sequence away; SWP swaps two values in the sequence; REV reverses the entire sequence; RHT shifts some elements in the sequence to the right (up); LFT shifts some elements in the sequence to the left (down).

10.1.18 Machine Rename_Nseq

This sequence machine provides access to a single sequence of numbers. It offers the common operations for Mathematical Concepts Machine:

- Getting a value (VAL)(VAL) with a particular ordinal number.
- Persistent Storage facilities (SAV, RST)

It offers the common sequence operation:

- Familiar Sequence Operations: (CLR, PSH, POP, FST, LST, TAL, STO) CLR clears the sequence; PSH pushes an element onto the sequence; POP gets the element most recently pushed; FST gets the first element; LST gets the last element; TAL modifies the sequence by removing the most recently pushed element; STO stores an element into the sequence.

- Some Sequence Enquiries: (EMP, LEN) EMP test whether the sequence is empty; LEN gives the length of the sequence.
- Some Sequence Tests: (EQL, NEQ) EQL tests whether a particular value is at a particular index; NEQ tests whether a particular value is not at a particular index.
- Additional Enquiries: (FUL, XST_IDX) FUL indicates whether a sequence is full; XST_IDX indicates whether a particular number is a valid index.
- Searching Operations: (SCH_LO_EQL, SCH_LO_NEQ, SCH_HI_EQL, SCH_HI_NEQ) The searching operation search for the lowest (or highest) index in the sequence which is equal (or not equal) to a particular value.
- Sequence manipulation operations: (KEP, CUT, SWP, REV, RHT, LFT) KEP keeps the beginning of a sequence; CUT cuts the beginning of a sequence away; SWP swaps two values in the sequence; REV reverses the entire sequence; RHT shifts some elements in the sequence to the right (up); LFT shifts some elements in the sequence to the left (down).

In addition the Number Sequence Machine offers:

- Extracting Values: (MAX_IDX, MIN_IDX) These operation extract the maximum (or minimum) value within a particular range.
- Arithmetic Operations: (ADD, MUL, SUB, DIV, MOD) These operations modify a particular location within the sequence.
- Test Operations: (GTR, GEQ, SMR, LEQ) These operations test a particular location within the sequence against a given value.
- Searching Operations: (SCH_LO_GEQ, SCH_LO_GTR, SCH_LO_LEQ, SCH_LO_SMR, CH_HI_GEQ, SCH_HI_GTR, SCH_HI_LEQ, SCH_HI_SMR) These operations search for the lowest index (or highest index) in which a value satisfying a particular condition can be found.
- Sequence manipulations: (SRT_AS, SRT_DSC) Sort the sequence to contains either ascending or descending values.

10.1.19 Machine Rename_Vfnc

This function machine provides access to a single partial function over numbers (mapping). The type of elements which the function returns can be determined through instantiation. It offers the common operations for Mathematical Concepts Machine:

 ©1996 *B-Core (UK) Ltd.*

- Getting a value by applying the function (VAL) .
- Persistent Storage facilities (SAV RES)

In addition it offers:

- Changing the function: (STO RMV) STO gives the function a new value for a particular field number; RMV removes a field number from the domain of the function.
- Testing function values: (EQL NEQ) These tests for a particular value in a particular field. in the function.
- Enquiries: (TST_FLD, DEF FREE) TST_FLD indicates whether a given number is a valid field number; DEF tests whether a given number is within the domain of the function; FREE return a field number for which no value has yet been given.

10.1.20 Machine Rename_Nfnc

This function machine provides access to a single partial function over numbers (mapping). The function will return numbers. It offers the common operations for Mathematical Concepts Machine:

- Getting a value by applying the function (VAL).
- Persistent Storage facilities (SAV, RES)

It offers:

- Changing the function: (STO, RMV) STO gives the function a new value for a particular field number; RMV removes a number from the domain of the function.
- Arithmetic operations: (ADD, MUL, SUB, DIV, MOD) These operation modifies the value of the function for a particular field number.
- Testing function values: (EQL, NEQ, GTR, GEQ, SMR, LEQ) These tests for a given value in a particular field in the function.
- Enquiries: (TST_FLD, DEF, FREE) TST_FLD indicates whether a particular number is a valid field number; DEF tests whether a particular number is within the domain of the function; FREE return a field number for which no value has yet been given.

 ©1996 *B-Core (UK) Ltd.*

10.1.21 Machine Rename_Vffnc

This function machine provides access to a single partial function over numbers (mapping). The type of elements which the function returns can be determined through instantiation. It offers the common operations for Mathematical Concepts Machine:

- Getting a value by applying the function (VAL).
- Persistent Storage facilities (SAV, RES)

It offers the additional operations:

- Changing the function: (STO, RMV) STO gives the function a new value fro a particular field number; RMV removes a number from the domain of the function.
- Testing function values: (EQL, NEQ) These tests for a given value in a particular field. in the function.
- Enquiries: (TST_FLD, DEF FREE) TST_FLD indicates whether a particular number is a valid field number; DEF tests whether a particular number is within the domain of the function; FREE return a field number for which no value has yet been given.

Some special operation for manipulation of segments of fields:

- Moving segments of values: (MOV) This operation moves the values stored in a seqment of fields to another segment of fields.
- String Operations: (OVR, XTR) OVR over-writes a number of function fields with a packed string; XTR extracts a string from a number of function fields.
- String Testing: (EQL) Tests whether a particular string is equal to a string packed in a number of fields.

10.1.22 Input Output Machines

The Input Output Machines offers an application some simple input and output facilities:

- basic_io, for basic terminal IO.
- Rename_token_io, for more terminal IO.
- Rename_file_io, for general file IO.
- file_dump, for persistent storage.

 ©1996 *B-Core (UK) Ltd.*

10.1.23 Machine basic_io

This machine provides operations for writing to the terminal display and reading from the keyboard:

- Spacing operations: (NWL, BLK)
- Writing and reading numbers: (GET_NBR, GET_PROMPT_NBR, PUT_NBR) An interval to which the numbers must belong is specified.
- Writing and reading B numbers: (GET_NAT, GET_PROMPT_NAT, PUT_NAT)
- Writing and reading boolean values: (GET_BOOL, GET_PROMPT_BOOL, PUT_BOOL)
- Writing and reading binary values: (GET_BTS, GET_PROMPT_BTS, PUT_BTS)
- Writing and reading strings: (GET_STR, GET_PROMPT_STR, PUT_STR)
- Writing and reading characters: (GET_CHR, PUT_CHR)

10.1.24 Machine Rename_token_io

This IO facility allows for a value of any elementary type to be read from the keyboard and written to the screen. **Note**This includes SEQOBJ, SETOBJ etc., since all these are represented as a single machine word) We have: (Rename_GET_TOK, Rename_GET_PROMPT_TOK, Rename_PUT_TOK)

10.1.25 Machine Rename_file_io

A new file can be introduced by importing an instance of the file_io machine. This new file can be opened for writing by *Rename_open_write_file*. All subsequent 'saves' will now be written to that file. The file can be opened for reading by *Rename_open_read_file*. All subsequent 'restores' will now be read from that file. *Rename_close_file* will close the file. A file should always be closed to ensure that all written information is properly recorded. Only one file can be open at any one time.

10.1.26 Machine file_dump

This machine provides the default dump file (TMP.file), which is known by all object machines. Unless otherwise specified this file can be used for 'saving' and 'restoring'. *write_dump_file* will open the dump file for saving. *open_read_dump_file* will open the dump file for restoring. *close_dump_file* will close the dumpfile. A file should always be closed to ensure that all written information is properly recorded.

 ©1996 *B-Core (UK) Ltd.*

10.1.27 Type Machines

The following machines allow for the manipulation of local variables introduced in an algorithm within an implementation. A value might be retrieved from the state of an object machine into a local variable, and then modified using the operation provided by the Type Operation Machines before stored into an object machine.

10.1.28 Machine Bool_TYPE

Introduces the type BOOL which consists only of the values TRUE and FALSE. The Bool machine provides the familiar boolean operations:

- Conjunction (and): CNJ
- Disjunction (or): DIS
- Negation (not): NEG
- Bit conversion: BTS This operation returns 1 for input TRUE and 0 for FALSE input .

10.1.29 Machine String_TYPE

This machine introduces the types CHAR and STRING. The operation CPY_STR must be used for assigning a value of type string to a local variable.

10.1.30 Machine Scalar_TYPE

This machine introduces the type of SCALAR (= 0..2147483646). The following operations are provided:

- Assigning a Scalar SCL
- Arithmetic Operations (INC, DEC, ADD, MUL, SUB, DIV, MOD)
- Testing operations: (EQL, NEQ, GTR, GEQ, SMR, LEQ, EQZ) EQZ tests for zero.

Note that the normal syntax available in programming languages can be used within an algorithm of a B implementation. e.g ‘ :=’ for assignment, ‘ +’ for ADD, ‘ =’ for EQL. However, if this syntax is used then no checks will be made by the B-Toolkit for over-flow.

 ©1996 *B-Core (UK) Ltd.*

10.1.31 Machine Bit_TYPE

This machine introduces the type of BITS as a set of sequences of length 32. The most significant bit position is 1, the least significant position is 32. We have:

- Bit operations: (VAL, STO) VAL returns the value of a bit; STO stores a new bit.
- Shifts: (LFT, RHT) LFT shifts left; RHT shifts right (no wrap-around).
- Bit-wise Logical Operations: (CPL, LND, LOR, LXR) CPL negates all bits; LND performs a bit-wise 'and'; LOR performs bit-wise 'or'; LXR performs bit-wise 'exclusive or'.
- Masking Operations: (MSK, VMS, MMS) MSK generates a mask of 1's; VMS performs a masking and returns the resulting value; MMS merges a pattern with another pattern using a mask.

 ©1996 *B-Core (UK) Ltd.*

10.2 The Team Library (TLIB)

This facility parallels that of the System Library (SLIB) and enables members of a team to create a library into which they may export a (sub) development, and from which they may import a development.

10.2.1 Making a New Project Team Library

1. Create an empty directory (with appropriate name) (e.g. some-path/BTeamLib); all team members must have read/write permission to the directory.
2. Make the directory into a B development directory by invoking the B-Toolkit from within the new directory. Leave the development empty.
3. The team member who will be working on sub-developments within the team must set the environment variable BTEAMLIB: e.g.

```
setenv BTEAMLIB some-path/BTeamLib
```

The team library is now ready for use.

10.2.2 Exporting Developments to the Team Library:

This facility is available under Utilities, enabling a user to export a completed (sub) development from their directory to that of the current team library; the BTEAMLIB environment variable must be set as described above.

A completed development is a single MACHINE for which an IMPLEMENTATION is written and a binary code module exists (see Translator for getting binary code modules from IMPLEMENTATIONS).

A warning will be given if the MACHINE SEES other MACHINES. This can be ignored unless the intention is that the seen machine should be a non-sharable module private to the exported sub-development, in which case the seen module should be IMPORTED into the sub-development that is being exported.

Exported developments appear as library machines within the team library. The library machines can be used directly within the team library or can be committed into other B developments within the same team.

 ©1996 *B-Core (UK) Ltd.*

10.2.3 Committing Developments from the Team Library:

TLIB constructs may be introduced to a development through Introduce. If a TLIB construct SEES other MACHINES, a warning will be issued that these constructs should subsequently be introduced into the importing environment.

Each committed TLIB MACHINE is, of course, animatable.

©1996 *B-Core (UK) Ltd.*

10.3 CreateSLIB

This option is invoked from the Tool Bar, and facilitates the creation of a new library construct in the system library directory - SLIB . A library construct is a specification together with a code module, and this facility assists in the creation of the C code files.

The option is applicable to any analysed specification which is not already present in SLIB; the construct should, however, not SEE any non-renamable SLIB machines.

Two types of library construct may be created:

1. Create SLIB-Committable construct
2. Create lower-level SLIB construct

The former enables the created construct to be subsequently introduced into other developments, whereas the latter provides a library construct on which other SLIB constructs may be built (by importing - see below); this one, however, is not made available for direct introduction into other developments.

Importation of lower-level SLIB constructs is achieved through the LIBDEF (.ldf) file - a template is provided, and the IMPORTS clause should be completed as it would in an Abstract Machine Implementation (with parameters set where appropriate); if nothing is being imported, the clause should be removed. Lower level SLIB constructs may themselves import other lower-level constructs.

After the LIBDEF file has been edited, the following prompt appears:

```
Create code template files in SRC
```

Three templates are provided in editors to produce the '.h', '.g' and '.c' files that constitute a library code module. The first contains header information for the code modules of machines which SEE or IMPORT this construct together with the definitions of any operations that are to be in-lined. The second contains module parameterisation information to be used at link-time when the values of module parameters are known. The last file contains the C code of the operations not in-lined in the '.h' file.

Data should be declared in the '.g' file, and declared as 'extern' in the .h file. It is recommended that one of the existing SLIB code modules is inspected for guidance (for example 'Rename_Nvar').

As much information as possible will have been written to these three files (templates for operations in both the '.h' and '.c' file, and parameter information in the '.g' file).

The prompt:

 ©1996 *B-Core (UK) Ltd.*

is then offered which, when selected, attempts to create the library construct by first compiling the code provided in the three files. If compilation fails the reasons are given in the Display Area and the prompt re-appears. When the compilation succeeds, files are copied from the current ANL CFG TYP and CDE/C directories into the corresponding SLIB directories (so you must have write permission in that directory).

'Cancel' aborts the creation, but preserves the code files in the SRC directory.

 ©1996 *B-Core (UK) Ltd.*

10.4 ExportTLIB

Export is available under Utilities on the Tool Bar, and allows a developer to export a (sub) development - a MACHINE for which an IMPLEMENTATION is written and a binary code module exists.

For a full description see Team Library.

The environment variable BTEAMLIB must be set.

©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

B-TOOLKIT USERS' MANUAL

SECTION 11

Utilities

©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

11 Utilities

11.1 Reset

There are two ways of invoking the Reset command.

1. From the Constructs Area of the Main Environment. This allows an individual construct to be reset (see ResetConstruct).
2. From the Utilities tool on the Tool Bar, which allows the user to reset all constructs, all documents or all code:

   - Reset all constructs
   - Reset all documents
   - Reset all code

   In the last two options only the processed documents or code are reset; no other changes are made. Note that code corresponding to a library machine, or that which has been generated through an enumeration is not reset by the last option.

Resetting all constructs places the system under development in a state in which previously committed constructs remain committed but no further processing has been done, and so all the constructs require re-analysis etc.. However, knowledge of the state of the system prior to the reset is retained and the system can be restored automatically (after any editing) to that state by use of Remake. All library machines are retained, but all generated machines are removed (although the Analyser, BaseGen, Enumerator and Interface tools will participate in the remake to regenerate/reproduce the constructs). All code is also removed, with the exception of that of the library machines.

Thus a Reset all constructs immediately followed by a Remake will result in the system restored (at least) to its present state.

 ©1996 *B-Core (UK) Ltd.*

11.2 Find

Find is invoked from the Tool Bar, and is a facility which interrogates the files in the TYP directory to enable the origin and type of a variable, set, constant or operation name, together with a list of referencing constructs. It results in the prompt:

```
Identifier(s) to find:
```

A response of 'some_op', for example, might result in the following display in the Text Window:

```
some_op:

  OPERATION in fifi

   Input  - inp_var : NAT

   Output - out1_var : NAT
            out2_var : NAT

   Referenced:
     mimi
     gigi
```

indicating that the operation 'some_op' is an operation name declared in the construct 'fifi' having a single input parameter 'inp_var' of type NAT, and two output variables 'out1_var' and 'out2_var' both having type NAT; 'some_op' is referenced in the two constructs 'mimi' and 'gigi'.

Note that Find is applicable only to constructs which are currently analysed.

 ©1996 *B-Core (UK) Ltd.*

11.3 Palettes

The palettes are available on the Top Bar, and provide templates for machines, refinements and implementations, and lists of the types of Abstract Machine operations available for each of these constructs:

- Machines
- Machines Ops
- Refinements
- Refinement Ops
- Implementations
- Implementation Ops

Templates are also provided for:

- Bases
- Documents
- Enumerations
- Interfaces

©1996 *B-Core (UK) Ltd.*

11.4 SaveDevelopment

The SaveDevelopment facility is available under the Utilities tool and provides a means of saving the current state of the development.

A prompt for a directory name is given, and the directory is created in the current development directory (and so should not already exist - an error results in this case), and then a reset version of the development is written to the newly-created directory.

The directory may subsequently be viewed as a normal development directory in its own right; it is relocatable, and when the B-Toolkit is invoked, Remake will restore it to the state immediately prior to which it was saved.

The contents of the directory thus represents the minimal information required for a system rebuild; thus it may be regarded as a candidate for archiving, compression, incorporation into software control systems, and so on.

©1996 *B-Core (UK) Ltd.*

11.5 BUnlock

The B-Toolkit is a single-user system, and when the toolkit is currently being used the file `.BToolkitlock` is created, preventing other users from simultaneously using the toolkit in that directory. A system error could result in the file being created and the toolkit failing to run, and the BUnlock utility results in the removal of this file.

The script resides in the $BKIT directory and is, of course, invoked outside the B-Toolkit.

©1996 *B-Core (UK) Ltd.*

11.6 BProcessInfo

System/Window crashes etc. may result in B processes left 'hanging', which may, in turn, affect the ability to run the B-Toolkit because of the licensing mechanism. Running the script '"BProcessInfo' will display the user's B processes currently running (as well as those of other users) together with an invitation to kill the user's processes.

The script resides in the $BKIT directory and is, of course, invoked outside the B-Toolkit.

©1996 *B-Core (UK) Ltd.*

B-TOOLKIT USERS' MANUAL

SECTION 12

References and Index

©1996 *B-Core (UK) Ltd.*

©1996 *B-Core (UK) Ltd.*

12 References and Index

References

[1] Abrial, J-R., *B-Tool Reference Manual*, Edinburgh Portable Compiler, 1991

[2] Abrial, J-R., *The B-Book - Assigning Programs to Meanings*, Cambridge University Press, ISBN 0-521-49619-5, 926 pages (to appear)

[3] Abrial, J-R. et al, B-Technology Technical Overview, B-Core(UK) Ltd. 1993. (available from B-Core(UK) on request)

[4] Abrial, J-R., A Formal Approach to Large Software Construction, in van de Snepscheut, J.L.A (editor) Mathematics of Program Construction, Springer-Verlag, Berlin, 1989

[5] Abrial, J-R., On Constructing Large Software Systems, in van Leeuwen,J (editor), Algorithms, Software,Architecture. Information Processing 92, Vol. 1, Elsevier Science Publishers B.V., North Holland..

[6] Abrial, J-R., A Refinement Case Study (Using the Abstract Machine Notation), in *Proceedings of the Fourth Refinement Workshop*, Springer-Verlag, 1991.

[7] B-Core (UK) Ltd., *B-Technology, Technical Overview*

[8] Bicarregui, J., Richie, B., Invariants, Frames and Postconditions: a Comparison of the VDM and B Notations, in *FME'93: Industrial-Strength Formal Methods*, Springer-Verlag, 1993.

[9] Bicarregui, J., Dick, J., Matthews, B. and Woods, E., *"Quantitative Analysis of an Application of Formal Methods"*, in proc. Formal Methods Europe'96, LNCS 1051, Springer, Oxford, March 1996.

[10] Bicarregui, J., Dick, J., Matthews, B. and Woods, E., *"Making the most of formal specification through Animation, Testing and Proof"*, to appear in the Science of Computer Programming, 1996.

[11] Blamey, W., McNeil, I., *Productivity Gains Through the Use of the B-Method*, ITRU/273, BP Internal Report, 1991.

[12] Brown, N. and Mery, D., *A proof Environment for Concurrent Programs*, in *FME'93: Industrial-Strength Formal Methods*, Springer-Verlag, 1993.

[13] Carnot, M., DaSilva, C., Dehbonei, B., Mejia, F., Error-free Software Development for Critical Systems Using the B-Methodology, in *Proceedings of the Third IEEE International Conference on Software Reliability Engineering*, October 92, North Carolina, USA.

 ©1996 *B-Core (UK) Ltd.*

[14] Chapront, P. Vital Coded Processor and Safety Related Software Design, in Frey,H.H. (editor). Safety of Computer Control Systems 1992 (SAFECOMP 92), Computer System in Safety Critical Applications, Proceedings of the IFAC Symposium, Switzerland. Pergamon Press, 1992.

[15] DaSilva, C., Dehbonei, B., Mejia, F., Formal Specification in the Development of Industrial Applications: The Subway Speed Control Mechanism, in *Proceedings of the Fifth International Conference on Formal Description Techniques, FORTE'92*, October 92, Lannion, France.

[16] Dijkstra, E.W., *A Discipline of Programming*, Prentice Hall, 1976.

[17] He, J. et al, *Data Refinement Refined*, PRG Oxford,1985

[18] Hoare, J., Dick, J., Neilson D. and Sørenson, I., *"Applying the B Technologies to CICS"*, in proc. Formal Methods Europe'96, LNCS 1051, Springer, Oxford, March 1996.

[19] Jones, C.B., *Software Development. A Rigorous Approach*, Prentice Hall, 1980.

[20] Jones, C.B., *Systematic Software Development Using VDM* Prentice Hall, Englewood Cliffs, NJ 1986.

[21] Lano, K., *The B Language and Method - a guide to practical formal development*, Springer-Verlag, ISBN 3-540-76033-4, 236 pages

[22] Lano, K., Haughton, H., *Specification in B: an Introduction using the B-Toolkit*, Imperial College Press, ISBN 1-86094-008-0 (paperback: 1-86094-018-8), 250 pages

[23] Lee, M.K.O., Scharbach, P.N., Sørensen, I.H., Engineering Real Software Using Formal Methods, in *Proceedings of the Fourth Refinement Workshop*, Springer-Verlag, 1991.

[24] Morgan, C.C., *Programming From Specifications*, Prentice Hall, 1990.

[25] Spivey, J.M., *Understanding Z*, Cambridge University Press, 1988.

[26] Wordsworth, J., *Software Engineering with B*, Addison Wesley Longman, ISBN 0-201-40356-0, 352 pages

[27] Zermelo, E., *Untersuchung über die Grundlagen der Me genlehre*, 1908.

[28] Zermelo, E., *Neuer Beweis für die Möglichkeit ei er Wohlordnung der Mengenlehre*

 ©1996 *B-Core (UK) Ltd.*

Index

064;b-core.com, *3-17*

Abstract Machines, *1-8*, *1-9*, *1-11*, *3-3*, **3-7**, *4-16*, *5-6*, *7-3*, *7-9*, *8-6*, *10-25*
act param clause, **3-16**
act param list clause, *3-15*, **3-16**
ADD, *10-21*
AMN, *1-8*, **3-3**, *3-30*, *6-3*
  Operations, *3-3*, **3-7**, *3-19*, *3-44*, *11-5*
  Postconditions, *3-19*
  Substitutions, **3-3**, *3-3*, *3-16*
Analysed Form, *4-4*, **5-3**
Analyser, *1-8*, *1-14–1-16*, *4-7–4-9*, *4-11*, *4-13*, *4-16*, **5-3**, *5-6*, *5-12*, *5-13*, *6-19*, *7-9*, *7-11*, *8-3*, *9-9*, *10-25*, *11-3*, *11-4*
Animator, *1-8*, *1-15*, *3-37*, *5-3*, **5-6**, *5-12*
Annotations, **9-3**
  Positioning, **9-4**
Antecedent, *6-17*
Antecedents, **6-4**, *6-5–6-8*
ASCII, **3-40**
ASSERTIONS Clause, *1-28*, **3-11**, *6-19*, *9-4*, *9-7*
AutoProver, *1-8*, *1-11*, *1-15*, *1-23*, *1-27*, *5-3*, **6-21**, *6-23–6-25*, *6-27*, *6-30*, *6-31*

B-Method, *1-8*, **2-3**, *3-30*, *3-37*
B-Platform, *1-5*, *1-17*, *3-7*, *5-10*, **6-3**, *6-30*
  Memory, **6-18**
  Recompiling, *1-17*, **1-23**, *6-18*
B-Theory Language, **6-3**, *6-3*, *6-12*, *6-31*
B-Toolkit, *1-3*, **1-8**, *1-17*, *3-37*, *6-3*, *11-7*, *11-8*
  Installation, **1-17**, *1-17*
  Library Requirements, *1-17*, **1-21**
  Running, *1-17*, **1-21**
Backtracking, **6-25**
BASE, *1-15*, **7-6**
Base Generators, *1-8*, *1-11*, *1-15*, *1-16*, *5-12*, **7-3**, *9-3*, *9-4*, *11-3*
basic io, *10-19*, **10-20**
Bit TYPE, **10-22**
BLK, *10-20*
bool, **3-6**
Bool TYPE, **10-21**
BProcessInfo, *1-17*, **11-8**
BTSCNJ BOOL, *10-21*
BUnlock, *1-17*, **11-7**

Cartesian Product, **3-21**, *3-41*
Close, *1-14*, *4-7*, **4-8**, *4-9*, *5-3*
CNJ BOOL, *10-21*
Code Sharing, *3-34*, **3-34**
Comma Separator, *3-19*, **3-22**
Command Bar, *1-5*, **1-10**
Command Buttons, **1-11**, *4-4*
Commit, *1-6*, *1-8*, *1-13*, *1-14*, *4-7*, *4-8*, **4-9**, *5-3*, *5-12*, *5-13*, *11-3*
CommitEdits, *4-7–4-9*, *5-13*
Concatenation
  Generalised, **3-28**, *3-44*
Conjunction, **3-19**, *3-19*, *3-40*
Consequent, **6-4**, *6-5–6-7*
CONSTANTS Clause, *3-31*, *3-35*, *4-14*, *4-15*, *5-6*, *8-4*, *9-4*
CONSTANTS clause, **3-10**
CONSTRAINTS Clause, **3-8**, *6-19*, *9-4*
Constructs
  Area, *1-5*, *1-6*, *1-10*, *1-11*, **1-11**, *1-14–1-16*, *4-3*, *4-4*, *4-6*, *4-12*, *5-13*, *6-20*, *6-22*, *9-3*, *9-8*, *11-3*, *11-4*
  Colour-coding, **1-11**, *4-4*
  Name length, **4-3**
CONTENTS, *3-17*
CPL BTS, *10-22*
CPY STR, *10-21*
CreateSLIB, *1-14*, **10-25**

DEC, *10-21*

©1996 *B-Core (UK) Ltd.*

DEFINITIONS Clause, **3-11**, *6-3*, *9-4*
Design Overview, *3-39*, **4-16**
Development Environment, *1-3*, **1-9**, *4-3, 4-4*, *4-6*, *4-9*, *4-12*, *4-14, 4-15*, *5-4*, *5-6*, *5-13*, *6-19*, *6-22*, *6-24*, *6-28*, *7-3*, *7-9*, *8-3*, *8-6*, *9-3*, *10-26*, *11-4*
DISCNJ BOOL, *10-21*
Disjunction, *3-19*, **3-20**, *3-40*
Disk Contents, *1-17*, **1-17**
Display Area, *1-5*, *1-10*, **1-14**, *4-10*, *4-12*, *7-14*, *10-26*
DIV, *10-21*
Document Mark Up, *1-8, 1-9*, *1-12*, *1-16*, *1-27, 1-28*, *4-11*, *5-3*, *6-31*, **9-3**, *9-8–9-11*
Documents, *1-11*, *6-31*, **9-8**

Editor, *1-8, 1-9*, *1-14*, *1-24*, *4-3*, **4-6**, *4-8, 4-9*, *4-11*, *4-16*, *5-13*
enumerated set clause, *3-10*, *3-12*, *3-15*, **3-16**
Enumerator, *1-8*, *1-11*, *1-15, 1-16*, *5-4*, *5-12*, **7-9**, *7-12*, *11-3*
Environment, *1-21*, **1-24**
  Variables, *1-3*, **1-24**
    BCC, *1-9*, **1-25**, *8-3*, *8-6*
    BEDIT, *1-9*, **1-24**, *4-6*
    BHTML, **1-24**
    BHYPEDIT, **1-24**, *6-25*
    BKIT, *1-3*, *1-9*, **1-24**, *11-7, 11-8*
    BLIC, *1-9*, **1-24**
    BNOBELL, **1-24**
    BPDVI, *1-9*, **1-25**, *9-11*
    BSDVI, *1-9*, **1-25**, *9-10*
    BSITECODE, *1-3*, *1-9*, *1-17*, **1-24**
    BTEAMLIB, **1-25**, *4-5*, *10-23*, *10-27*
    PATH, **1-26**, *9-3*
    XDVIFONTS, **1-26**, *9-10*
  XBMotif, *1-6*, *1-10*, **1-26**
Environments, *1-6*, *1-10, 1-11*, **1-14**, *4-4*
  Documents, *1-11*, **1-16**, *6-31*, *9-3*, *9-8*, *9-10, 9-11*
  Generators, *1-11*, **1-15**, *4-11, 4-12*, *5-6*, *7-3*, *7-9*, *7-11*, *11-3*
  Main, *1-5*, *1-11*, **1-15**, *4-13*, *5-3*, *5-6*, *6-19*, *11-3*
  Provers, *1-11*, **1-15**, *6-20–6-24*, *6-28*, *6-31–6-33*
  Translators, *1-11*, **1-16**, *7-13*, *8-3*, *8-6, 8-7*
EQL, *10-21*
Equals, *3-19*, **3-20**, *3-40*
Equivalence, *3-19*, **3-20**, *3-40*
EQZ, *10-21*
ExportTLIB, *1-14*, *4-4*, **10-27**
Expression List, **3-20**
Expressions, *3-18*, **3-20**
EXTENDS Clause
  Syntax, **3-10**, *3-32*
  Using, **3-32**, *9-4*

FALSE, *10-21*
file dump, *10-19*, **10-20**
Find, *1-13*, *5-5*, **11-4**
Formula, *3-7, 3-8*, *3-10–3-13*, *3-15, 3-16*, **6-3**
Freeness, **3-29**, *3-45*, *6-10*, *6-13*
Functions, *3-18*, **3-26**
  Application, **3-27**, *3-43*
  Bijection, *3-19*, **3-27**, *3-40*
  Partial, *3-19*, **3-26**, *3-40*
  Total, *3-19*, **3-26**, *3-40*, *6-9*, *6-14*

Generalised Substitution Language, **2-5**
GEQ, *10-21*
GET BOOL, *10-20*
GET BTS, *10-20*
GET CHR, *10-20*
GET NAT, *10-20*
GET NBR, *10-20*
GET PROMPT BOOL, *10-20*
GET PROMPT BTS, *10-20*
GET PROMPT NAT, *10-20*

 ©1996 *B-Core (UK) Ltd.*

GET PROMPT NBR, *10-20*
GET PROMPT STR, *10-20*
GET STR, *10-20*
Getting Started, **1-3**, *1-22*
GLOBAL, **7-6**
Goals
  Current, **6-14**, *6-15*, *6-24*
  Set, **6-14**
GTR, *10-21*
Guards, **6-8**, *6-17*
  bident, *5-10*, **6-11**
  binhyp, **6-9**
  breade, **6-10**
  bsearch, **6-10**
  btest, *5-11*, **6-9**, *6-16*

Help, **1-6**, *1-6*
Hypertext, *1-9*, *1-14*, *1-24*, **4-14**
Hypotheses
  Current set, *6-7*, *6-13, 6-14*, **6-14**, *6-15*, *6-17*, *6-25*
  Viewing, *1-24*, **6-25**

Identifier, *3-7*, *3-10–3-16*, **3-28**, *6-4*, *6-12*, *7-5*, *7-11*
  Upper Case, **3-28**
  Variable, **3-28**
  Variable List, **3-28**
IMPLEMENTATION, *1-11*, *1-16*, *3-3*, **3-14**, *3-30*, *3-32*, *3-34*, *3-37*, *4-13*, *4-14*, *4-16*, *5-3*, *6-21*, *6-23*, *6-31, 6-32*, *7-11*, *8-3*, *8-6*, *9-4*, *10-23*, *10-25*, *10-27*, *11-5*
Implication, **3-19**, *3-19*, *3-40*
IMPORTS Clause, *3-37*, *3-39*
  Syntax, *3-9*, **3-15**, *3-34*
  Using, *3-30*, **3-34**, *4-13*, *4-16*, *5-4*, *7-11*, *8-3, 8-4*, *9-4*, *10-3*, *10-23*, *10-25*
INC, *10-21*
INCLUDES Clause
  Syntax, **3-9**, *3-31*
  Using, *3-30*, **3-31**, *4-13*, *4-16*, *5-4*, *9-4*
Inference
  Backwards, *6-3*, **6-5**, *6-6*, *6-13*, *6-15*, *6-17*
  Forwards, *6-3*, **6-7**, *6-13*, *6-15*, *6-17*
  Rewriting, *6-3*, **6-6**
Inference Rules, *3-7*, **6-3**, *6-12*
  Backward, *6-4*, **6-7**
  Forward, *6-4*, **6-6**, *6-17*
  Ordering, *6-12*, **6-16**, *6-17*
  Rewrite, *3-19*, *3-41*, *5-10*, *6-4*, **6-5**, *6-6, 6-7*, *6-16, 6-17*, *6-29*
  Simple, *6-4*, **6-4**, *6-7*, *6-17*
Information Hiding, **3-30**
Information Panel, *1-3*, *1-10*, **1-13**
INITIALISATION Clause, *1-28*, **3-11**, *8-4*, *9-4*, *9-7*
Injection
  Partial, *3-19*, **3-26**, *3-40*
  Total, *3-19*, **3-26**, *3-40*
Installation, *1-9*, **1-17**
  Errors, **1-22**
Interface
  Generator, *1-8*, *1-11, 1-12*, *1-15, 1-16*, *1-27*, *1-29*, *4-3*, **7-11**, *11-3*
    Motif, *1-16*, **7-13**, *8-7*
    Non Motif, *4-14*, *7-10*, **7-12**, *7-13*
  Motif, *1-3*, **1-10**, *1-27*, *4-4*, *6-32*
Interface Generator, *3-39*
InterProver, *1-8*, *1-15*, *6-12, 6-13*, *6-21, 6-22*, **6-23**
Introduce, *1-6*, *1-13*, **4-3**, *4-12*, *7-11*, *8-3*, *9-8*, *10-24, 10-25*
IntroduceSLIB, **4-4**, *4-14*
IntroduceSRC, **4-4**
IntroduceTLIB, **4-4**, *4-14*
INVARIANT Clause, *1-28*, **3-10**, *5-5*, *6-19*, *9-4*, *9-7*

Joker, **6-3**, *6-8*

LaTeX, *1-28*, *6-31*, **9-3**, *9-6*
Layered Approach, *3-34*, **3-37**
Lemma, **6-24**

©1996 *B-Core (UK) Ltd.*

Lemma creation, *6-23*, **6-24**
Lemma proof, *6-23, 6-24*, **6-27**
LEQ, *10-21*
LFT BTS, *10-22*
Library Machines, **10-4**
  Input Output Machines, *10-4*, **10-19**
  Mathematical Concepts Machines, *10-4*, **10-14**
  Multiple Objects Machines, **10-4**, *10-4*
  Programming Concepts Machines, *10-4*, **10-10**
  Type Machines, *10-4*, **10-20**
Licence File, *1-17*, **1-21**, *11-8*
Linker, *1-8*, *1-12*, *1-16*, *1-27*, *3-36*, *8-3, 8-4*, **8-6**, *8-7*, *10-25*
  Name Clashes, **3-36**
LND BTS, *10-22*
Lock, **11-7**
LOR BTS, *10-22*
LXR BTS, *10-22*

MACHINE, *2-7*, **3-8**, *3-31*, *4-13, 4-14*, *5-3*, *6-21*, *6-23*, *6-31, 6-32*, *7-11*, *9-4*, *10-23*, *10-27*, *11-5*
machine ref clause, *3-12*, *3-14, 3-15*, **3-16**
Mathematical Notation, **3-18**, *6-3*
MiniRemake, *4-7–4-9*, **5-12**, *6-19*, *7-8*
MMS BTS, *10-22*
MOD, *10-21*
Mouse, **1-5**
MSK BTS, *10-22*
MUL, *10-21*

Natural Numbers, *3-18*, **3-22**
  Addition, *3-19*, **3-23**, *3-42*
  Difference, *3-19*, **3-23**, *3-42*
  Division, *3-19*, **3-23**, *3-42*
  Inequality, *3-19*, **3-23**, *3-42*
  Interval, *3-19*, **3-23**, *3-42*
  Maximum, **3-23**, *3-42*
  Minimum, **3-23**, *3-42*
  Product, *3-19*, **3-23**, *3-42*
  Remainder, *3-19*, **3-23**, *3-42*
  Set of, **3-23**, *3-42*
    Non zero, **3-23**, *3-42*
Negation, **3-19**, *3-40*
  Not Equals, *3-19*, **3-20**, *3-40*
  Not Set Inclusion, *3-19*, **3-21**, *3-41*
  Not Set Member, *3-19*, **3-20**, *3-41*
  Not Strict Set Inclusion, *3-19*, **3-21**
NEGCNJ BOOL, *10-21*
NEQ, *10-21*
Newlines, **9-4**
Non Determinism, **3-6**, *3-33*
Normaliser, **5-3**
Number, *3-7*, *3-16*, *6-9*, **6-11**
NWL, *10-20*

ON Clause, *6-21*, **6-29**
operation clause, *3-11*, *3-16*, **3-16**, *3-16*
operation ref clause, *3-15*, **3-16**
Operations
  Query, **3-32**
OPERATIONS Clause, **3-11**, *4-14, 4-15*, *8-5*, *9-4*
Operator Binding, **3-18**
Operator Priorities, *3-18*, **3-19**
Options, *1-6*, *1-11*, *1-15, 1-16*, **1-27**, *9-7*
  Construct Display, *1-12*, *1-15, 1-16*, **1-27**
  Document Mark Up, *9-9*
  Document Mark Up Language, *1-12*, **1-28**
  Documents, *1-12*, *1-28*, **9-7**
  Editor, *1-12*, **1-27**
  Interface, *1-12*, **1-29**, *7-12*
  Programming Language, *1-12*, **1-28**, *8-3*
  Remake, *1-11*, **1-27**, *5-13*
Ordered Pair, *3-19*, **3-20**, *3-41*, *6-14*
Overview, *1-9*, *1-11*, *1-14*, *3-34*, *3-39*, **4-16**

Palette, *1-6*, *1-12*
Palettes, **11-5**

©1996 *B-Core (UK) Ltd.*

param list clause, *3-8–3-10*, *3-16*, **3-16**
Pattern Match, *6-5–6-7*, **6-7**, *6-16*
PMD DEFINITIONS Clause, **6-29**
PMD INCLUSIONS Clause—bold, 6-29
POGenerator, *4-7*
PRE Clause, **3-3**, *6-19*
Predicates, *3-18*, **3-19**
Print, *1-16*, *1-25*, **9-11**
PROMOTES Clause
  Syntax, **3-10**, *3-32*, *4-15*
  Using, **3-32**, *9-4*
Proof
  Construction, *1-23*, **6-14**
  Cycle, **6-21**
  Guidelines, **6-15**
  Levels, **6-21**, *6-31*, *6-33*
Proof Obligation, *2-8*, *2-12*, **6-19**, *6-21*
  File, **6-19**, *6-22*
  Generator, *1-8*, *1-11*, *1-14, 1-15*, *1-27*, *4-8, 4-9*, *5-3*, *5-12*, **6-19**, *6-21*, *6-23*, *6-31*
PROOFMETHOD File, *1-15*, *6-12, 6-13*, *6-21–6-25*, *6-27*, **6-28**, *6-28*, *6-30, 6-31*
ProofPrinter, *1-8*, *1-11*, *1-15*, *1-27*, **6-31**, *9-3*
PROPERTIES Clause, *1-28*, **3-10**, *3-35*, *5-4*, *6-19*, *8-4*, *9-4*, *9-7*
PUT BOOL, *10-20*
PUT BTS, *10-20*
PUT CHR, *10-20*
PUT NAT, *10-20*
PUT NBR, *10-20*
PUT STR, *10-20*

Quantification
  Existential, **3-20**, *3-40*, *6-11*
  Lambda Abstraction, **3-27**, *3-43*
  Unbounded Choice, **3-29**, *3-45*
  Universal, **3-20**, *3-40*, *6-14*
Quoted String, **6-11**

Reasoning Depth, *6-21*, *6-25*, **6-29**
REFINEMENT, *1-11*, *2-12*, *3-3*, **3-11**, *3-30*, *3-34*, *4-13, 4-14*, *4-16*, *5-3*, *6-21*, *6-23*, *6-31, 6-32*, *9-4*, *11-5*
Refinement, **3-30**
REFINES Clause, **3-12**, *4-13*, *5-4*, *9-4*
Relations, *3-18*, **3-24**
  Anti Domain Restriction, *3-19*, **3-25**, *3-43*
  Anti Range Restriction, *3-19*, **3-25**, *3-43*
  Closure, **3-26**, *3-43*
  Codomain, **3-24**, *3-42*
  Composition, *3-19*, **3-24**, *3-42*
  Domain, **3-24**, *3-42*
  Domain Restriction, *3-19*, **3-25**, *3-43*
  Identity, **3-24**, *3-42*
  Image, **3-25**, *3-43*
  Inverse, **3-25**, *3-43*
  Iteration, **3-26**, *3-43*
  Overriding, *3-19*, **3-25**, *3-43*, *6-14*
  Product, Direct, **3-25**
  Product, Parallel, **3-25**
  Projection, **3-26**, *3-43*
  Range Restriction, *3-19*, **3-25**, *3-43*
  Relation, *3-19*, **3-24**, *3-40*
Remake, *1-8, 1-9*, *1-14*, *1-27*, *4-7–4-9*, *5-3*, *5-12*, **5-13**, *6-33*, *11-3*, *11-6*
Remove, *1-14*, **4-12**
RemoveLevel, *1-15*
Rename, *1-14*, **4-11**
Rename ADD NARR, *10-13*
Rename ADD NFNC, *10-18*
Rename ADD NSEQ, *10-17*
Rename ADD NVAR, *10-12*
Rename ANY FNC OBJ, *10-9*
Rename ANY SEQ OBJ, *10-7*
Rename ANY SET OBJ, *10-6*
Rename ANY STR OBJ, *10-8*
Rename APP SEQ OBJ, *10-7*
Rename APP STR OBJ, *10-8*
Rename ASC NARR, *10-14*
Rename CH HI GEQ NSEQ, *10-17*
Rename CLR NSEQ, *10-16*

©1996 *B-Core (UK) Ltd.*

Rename CLR SEQ, *10-16*
Rename CLR SEQ OBJ, *10-7*
Rename CLR SET, *10-15*
Rename CLR SET OBJ, *10-6*
Rename CLR STR OBJ, *10-8*
Rename CPY SEQ OBJ, *10-7*
Rename CPY SET OBJ, *10-6*
Rename CPY STR OBJ, *10-8*
Rename CRD SET, *10-15*
Rename CRD SET OBJ, *10-6*
Rename CRE FNC OBJ, *10-9*
Rename CRE SEQ OBJ, *10-6*
Rename CRE SET OBJ, *10-6*
Rename CRE STR OBJ, *10-8*
Rename CUT NSEQ, *10-17*
Rename CUT SEQ, *10-16*
Rename CUT SEQ OBJ, *10-7*
Rename CUT STR OBJ, *10-8*
Rename DEC NVAR, *10-12*
Rename DEF FNC, *10-18, 10-19*
Rename DEF FNC FNC OBJ, *10-9*
Rename DEF FNC OBJ, *10-10*
Rename DEF NFNC, *10-18*
Rename DIFF SET OBJ, *10-6*
Rename DIV NARR, *10-13*
Rename DIV NFNC, *10-18*
Rename DIV NSEQ, *10-17*
Rename DIV NVAR, *10-12*
Rename EMP NSEQ, *10-17*
Rename EMP SEQ, *10-16*
Rename EMP SEQ OBJ, *10-7*
Rename EMP SET, *10-15*
Rename EMP SET OBJ, *10-6*
Rename EMP STR OBJ, *10-8*
Rename ENT SET, *10-15*
Rename ENT SET OBJ, *10-6*
Rename EQL ARR, *10-12*
Rename EQL FFNC, *10-19*
Rename EQL FFNC OBJ, *10-10*
Rename EQL FNC, *10-18, 10-19*
Rename EQL LIT STR OBJ, *10-8*
Rename EQL NARR, *10-13*
Rename EQL NFNC, *10-18*
Rename EQL NSEQ, *10-17*
Rename EQL NVAR, *10-12*
Rename EQL SEQ, *10-16*
Rename EQL SEQ OBJ, *10-7*
Rename EQL STR OBJ, *10-8*
Rename EQL VAR, *10-12*
Rename ffnc obj, *10-4*, **10-9**
Rename file io, *10-19*, **10-20**
Rename FIRST FNC OBJ, *10-9, 10-10*
Rename FIRST SEQ OBJ, *10-7*
Rename FIRST SET OBJ, *10-6*
Rename FIRST STR OBJ, *10-8*
Rename fnc obj, *10-4*, **10-9**
Rename FREE FNC, *10-18, 10-19*
Rename FREE NFNC, *10-18*
Rename FST NSEQ, *10-16*
Rename FST SEQ, *10-16*
Rename FUL FNC OBJ, *10-9*
Rename FUL NSEQ, *10-17*
Rename FUL SEQ, *10-16*
Rename FUL SET, *10-15*
Rename GEQ NARR, *10-13*
Rename GEQ NFNC, *10-18*
Rename GEQ NSEQ, *10-17*
Rename GEQ NVAR, *10-12*
Rename GET PROMPT TOK, *10-20*
Rename GET TOK, *10-20*
Rename GTR NARR, *10-13*
Rename GTR NFNC, *10-18*
Rename GTR NSEQ, *10-17*
Rename GTR NVAR, *10-12*
Rename INC NVAR, *10-12*
Rename INCL SET OBJ, *10-6*
Rename INI FNC OBJ, *10-9*
Rename INI SEQ OBJ, *10-6*
Rename INI SET OBJ, *10-6*
Rename INI STR OBJ, *10-8*
Rename INTER SET OBJ, *10-6*
Rename KEP NSEQ, *10-17*
Rename KEP SEQ, *10-16*
Rename KEP SEQ OBJ, *10-7*
Rename KEP STR OBJ, *10-8*
Rename KIL FNC OBJ, *10-9*

©1996 *B-Core (UK) Ltd.*

Rename KIL SEQ OBJ, *10-6*
Rename KIL SET OBJ, *10-6*
Rename KIL STR OBJ, *10-8*
Rename LEN NSEQ, *10-17*
Rename LEN SEQ, *10-16*
Rename LEN SEQ OBJ, *10-7*
Rename LEN STR OBJ, *10-8*
Rename LEQ NARR, *10-13*
Rename LEQ NFNC, *10-18*
Rename LEQ NSEQ, *10-17*
Rename LEQ NVAR, *10-12*
Rename LFT ARR, *10-13*
Rename LFT NARR, *10-14*
Rename LFT NSEQ, *10-17*
Rename LFT SEQ, *10-16*
Rename LST NSEQ, *10-16*
Rename LST SEQ, *10-16*
Rename MAX IDX NARR, *10-13*
Rename MAX IDX NSEQ, *10-17*
Rename MAX NVAR, *10-12*
Rename MBR SEQ OBJ, *10-7*
Rename MBR SET, *10-15*
Rename MBR SET OBJ, *10-6*
Rename MEM FUL SEQ OBJ, *10-7*
Rename MEM FUL SET OBJ, *10-6*
Rename MEM FUL STR OBJ, *10-8*
Rename MIN IDX NARR, *10-13*
Rename MIN IDX NSEQ, *10-17*
Rename MIN NVAR, *10-12*
Rename MOD NARR, *10-13*
Rename MOD NFNC, *10-18*
Rename MOD NSEQ, *10-17*
Rename MOD NVAR, *10-12*
Rename MOV FFNC, *10-19*
Rename MOV FFNC OBJ, *10-10*
Rename MUL NARR, *10-13*
Rename MUL NFNC, *10-18*
Rename MUL NSEQ, *10-17*
Rename MUL NVAR, *10-12*
Rename Narr, *10-10*, **10-13**
Rename NEQ ARR, *10-12*
Rename NEQ FNC, *10-18, 10-19*
Rename NEQ NARR, *10-13*
Rename NEQ NFNC, *10-18*
Rename NEQ NSEQ, *10-17*
Rename NEQ NVAR, *10-12*
Rename NEQ SEQ, *10-16*
Rename NEQ VAR, *10-12*
Rename NEW STR OBJ, *10-8*
Rename NEXT FNC OBJ, *10-9, 10-10*
Rename NEXT SEQ OBJ, *10-7*
Rename NEXT SET OBJ, *10-6*
Rename NEXT STR OBJ, *10-8*
Rename Nfnc, *10-14*, **10-18**
Rename Nseq, *10-14*, **10-16**
Rename Nvar, *10-10*, **10-12**
Rename OBJ FUL SEQ OBJ, *10-7*
Rename OBJ FUL SET OBJ, *10-6*
Rename OBJ FUL STR OBJ, *10-8*
Rename OVR FFNC, *10-19*
Rename OVR FFNC OBJ, *10-10*
Rename OVR SEQ OBJ, *10-7*
Rename OVR STR OBJ, *10-8*
Rename POP NSEQ, *10-16*
Rename POP SEQ, *10-16*
Rename POP SEQ OBJ, *10-7*
Rename POP STR OBJ, *10-8*
rename prefix clause, **3-11**
Rename PSH NSEQ, *10-16*
Rename PSH SEQ, *10-16*
Rename PSH SEQ OBJ, *10-7*
Rename PSH STR OBJ, *10-8*
Rename PUT TOK, *10-20*
Rename RES ARR, *10-13*
Rename RES FNC, *10-18, 10-19*
Rename RES NARR, *10-13*
Rename RES NFNC, *10-18*
Rename RES NVAR, *10-12*
Rename RES VAR, *10-12*
Rename REV ARR, *10-13*
Rename REV NARR, *10-14*
Rename REV NSEQ, *10-17*
Rename REV SEQ, *10-16*
Rename REV SEQ OBJ, *10-7*
Rename REV STR OBJ, *10-8*
Rename RHT ARR, *10-13*

 ©1996 *B-Core (UK) Ltd.*

Rename RHT NARR, *10-14*
Rename RHT NSEQ, *10-17*
Rename RHT SEQ, *10-16*
Rename RMV FNC, *10-18, 10-19*
Rename RMV FNC OBJ, *10-9*
Rename RMV NFNC, *10-18*
Rename RMV SET, *10-15*
Rename RMV SET OBJ, *10-6*
Rename RST FNC OBJ, *10-9, 10-10*
Rename RST NSEQ, *10-16*
Rename RST SEQ, *10-15*
Rename RST SEQ OBJ, *10-7*
Rename RST SET, *10-15*
Rename RST SET OBJ, *10-6*
Rename RST STR OBJ, *10-8*
Rename SAV ARR, *10-13*
Rename SAV FNC, *10-18, 10-19*
Rename SAV FNC OBJ, *10-9, 10-10*
Rename SAV NARR, *10-13*
Rename SAV NFNC, *10-18*
Rename SAV NSEQ, *10-16*
Rename SAV NVAR, *10-12*
Rename SAV SEQ, *10-15*
Rename SAV SEQ OBJ, *10-7*
Rename SAV SET, *10-15*
Rename SAV SET OBJ, *10-6*
Rename SAV STR OBJ, *10-8*
Rename SAV VAR, *10-12*
Rename SCH HI EQL ARR, *10-13*
Rename SCH HI EQL NARR, *10-14*
Rename SCH HI EQL NSEQ, *10-17*
Rename SCH HI EQL SEQ, *10-16*
Rename SCH HI GEQ NARR, *10-14*
Rename SCH HI GTR NARR, *10-14*
Rename SCH HI GTR NSEQ, *10-17*
Rename SCH HI LEQ NARR, *10-14*
Rename SCH HI LEQ NSEQ, *10-17*
Rename SCH HI NEQ ARR, *10-13*
Rename SCH HI NEQ NARR, *10-14*
Rename SCH HI NEQ NSEQ, *10-17*
Rename SCH HI NEQ SEQ, *10-16*
Rename SCH HI SMR NARR, *10-14*
Rename SCH HI SMR NSEQ, *10-17*
Rename SCH LO EQL ARR, *10-13*
Rename SCH LO EQL NARR, *10-14*
Rename SCH LO EQL NSEQ, *10-17*
Rename SCH LO EQL SEQ, *10-16*
Rename SCH LO GEQ NARR, *10-14*
Rename SCH LO GEQ NSEQ, *10-17*
Rename SCH LO GTR NARR, *10-14*
Rename SCH LO GTR NSEQ, *10-17*
Rename SCH LO LEQ NARR, *10-14*
Rename SCH LO LEQ NSEQ, *10-17*
Rename SCH LO NEQ ARR, *10-13*
Rename SCH LO NEQ NARR, *10-14*
Rename SCH LO NEQ NSEQ, *10-17*
Rename SCH LO NEQ SEQ, *10-16*
Rename SCH LO SMR NARR, *10-14*
Rename SCH LO SMR NSEQ, *10-17*
Rename seq obj, *10-4*, **10-6**
Rename set, *10-14*, **10-15**
Rename set obj, *10-4*, **10-5**
Rename SMR NARR, *10-13*
Rename SMR NFNC, *10-18*
Rename SMR NSEQ, *10-17*
Rename SMR NVAR, *10-12*
Rename SMR STR OBJ, *10-8*
Rename SRT AS NSEQ, *10-17*
Rename SRT DSC NARR, *10-14*
Rename SRT DSC NSEQ, *10-17*
Rename SRT NARR, *10-14*
Rename STO ARR, *10-12*
Rename STO FNC, *10-18, 10-19*
Rename STO FNC OBJ, *10-9*
Rename STO NARR, *10-13*
Rename STO NFNC, *10-18*
Rename STO NSEQ, *10-16*
Rename STO NVAR, *10-12*
Rename STO SEQ, *10-16*
Rename STO SEQ OBJ, *10-7*
Rename STO STR OBJ, *10-8*
Rename STO VAR, *10-11*
Rename str obj, *10-4*, **10-7**
Rename SUB NARR, *10-13*
Rename SUB NFNC, *10-18*
Rename SUB NSEQ, *10-17*

©1996 *B-Core (UK) Ltd.*

Rename SUB NVAR, *10-12*
Rename SWA SEQ OBJ, *10-7*
Rename SWA STR OBJ, *10-8*
Rename SWP ARR, *10-13*
Rename SWP NARR, *10-14*
Rename SWP NSEQ, *10-17*
Rename SWP SEQ, *10-16*
Rename TAL NSEQ, *10-16*
Rename TAL SEQ, *10-16*
Rename token io, *10-19*, **10-20**
Rename TST FLD FNC, *10-18, 10-19*
Rename TST FLD FNC OBJ, *10-9, 10-10*
Rename TST FLD NFNC, *10-18*
Rename TST IDX ARR, *10-13*
Rename TST IDX NARR, *10-14*
Rename UNION SET OBJ, *10-6*
Rename VAL ARR, *10-12*
Rename VAL FNC, *10-18, 10-19*
Rename VAL NARR, *10-13*
Rename VAL NFNC, *10-18*
Rename VAL NVAR, *10-12*
Rename VAL SEQ, *10-15, 10-16*
Rename VAL SEQ OBJ, *10-7*
Rename VAL SET, *10-15*
Rename VAL SET OBJ, *10-6*
Rename VAL STR OBJ, *10-8*
Rename VAL VAR, *10-11*
Rename Varr, *10-10*, **10-12**
Rename Vffnc, *10-14*, **10-19**
Rename Vfnc, *10-14*, **10-17**
Rename Vseq, *10-14*, **10-15**
Rename Vvar, *10-10*, **10-11**
Rename XST FNC OBJ, *10-9*
Rename XST IDX NSEQ, *10-17*
Rename XST IDX SEQ, *10-16*
Rename XST IDX SEQ OBJ, *10-7*
Rename XST IDX SET, *10-15*
Rename XST IDX STR OBJ, *10-8*
Rename XST ORD SET OBJ, *10-6*
Rename XST SEQ OBJ, *10-7*
Rename XST SET OBJ, *10-6*
Rename XST STR OBJ, *10-8*
Rename XTR FFNC, *10-19*
Rename XTR FFNC OBJ, *10-10*
Rename XTR STR OBJ, *10-8*
Renaming
    INCLUDES, **3-11**, *3-18, 3-19*, *3-31*
Renaming Library Machines, **10-3**
Reset, *1-15*, *4-10*, **11-3**
Reset Construct, *1-14*, **4-10**, *11-3*
ResetLevel, **6-33**
RHT BTS, *10-22*
rnm machine ref clause, *3-8–3-10*, **3-11**
rnm operation ref clause, *3-10*, **3-11**
Run, *1-16*, *7-13*, **8-7**

Save Development, *1-14*, **11-6**
Scalar TYPE, **10-21**
SCL, *10-21*
SEES Clause
    Syntax, **3-9**, *3-34*
    Using, *3-30*, **3-34**, *4-13*, *4-16*, *5-4*, *7-11*, *8-3, 8-4*, *8-6*, *9-4*, *10-3*, *10-23–10-25*
Sequences, *3-18*, **3-27**
    Append, *3-19*, **3-27**, *3-44*
    Bijective, **3-27**, *3-43*
    Concatenation, *3-19*, **3-27**, *3-43*, *5-10*
    Empty, **3-27**, *3-43*
    Enumeration, **3-28**, *3-44*
    Finite, **3-27**, *3-43*
    First Element, **3-28**, *3-44*
    Frint, **3-28**, *3-44*
    Injective, **3-27**, *3-43*
    Last Element, **3-28**, *3-44*
    Non empty, *3-43*
    Prepend, *3-19*, **3-27**, *3-44*
    Projection, *3-19*, **3-28**, *3-44*
    Reverse, **3-28**, *3-44*
    Singleton, **3-28**, *3-44*
    Size of, **3-28**, *3-44*
    Tail, **3-28**, *3-44*
set contents clause, **3-16**
SETS, *10-4, 10-5*, *10-11*
Sets, *3-18*, **3-20**
    Cardinality, **3-24**, *3-42*

©1996 *B-Core (UK) Ltd.*

Comprehension, **3-21**
Difference, **3-21**, *3-41*
Empty, **3-21**, *3-41*
Enumeration, **3-22**, *3-41*
Inclusion, *3-19*, **3-21**, *3-41*, *6-4*, *6-14*
Intersection, *3-19*, **3-21**, *3-41*
Generalised, **3-22**, *3-41*
UGeneralised, **3-22**, *3-41*
Membership, *3-19*, **3-20**, *3-41*
Power, **3-21**, *3-41*, *5-4*
Product, **3-24**, *3-42*
Singleton, **3-22**, *3-41*
Subsets
Finite, **3-22**, *3-41*
Finite Non Empty, **3-22**, *3-41*
Non empty, **3-22**, *3-41*
Summation, **3-24**, *3-42*
Union, *3-19*, **3-21**, *3-41*, *6-4*
Generalised, **3-22**, *3-41*
UGeneralised, **3-22**, *3-41*
SETS Clause, **3-10**, *3-31*, *4-14, 4-15*, *5-4*, *5-6*, *8-4*, *9-4*
Show, *1-16*, *1-25*, **9-10**
Show Unproved, *1-15*, **6-32**
Simple Rule, **6-4**
skip, **3-6**
SLIB, *3-37*
SMR, *10-21*
Specification Overview, *3-34*, **4-16**
Stateless Machine, **3-35**
Status, *1-8*, *1-15*, **4-13**
STO BTS, *10-22*
String TYPE, **10-21**
Structuring Developments, *2-9*, **3-30**
SUB, *10-21*
Substitutions
Application, *3-19*, **3-29**, *3-45*, *6-13*
Choice, *3-19*, **3-29**, *3-45*
Generalised, *1-8*, *3-3*, *3-18*, **3-29**, *6-13*
Guarded, *3-19*, **3-29**, *3-45*
Parameterised, **3-6**
Postconditioned, **3-6**, *3-44*
Preconditioned, *3-19*, **3-29**, *3-45*
Sequencing, *3-19*, **3-29**, *3-44*
Simple, **3-29**, *3-44*
Simultaneous, *3-19*, **3-29**, *3-44*, *6-8*, *6-11*
skip, **3-29**, *3-44*
Surjection
Partial, *3-19*, **3-26**, *3-40*
Total, *3-19*, **3-26**, *3-40*
System Library, *1-8*, *1-11*, *3-34*, *4-4*, *4-11, 4-12*, *4-14*, *5-4*, *5-6*, *7-3*, *7-8*, *7-12*, *8-3–8-5*, **10-3**, *10-23*, *10-25*, *11-3*
Files
.c, **10-25**
.g, **10-25**
.h, **10-25**
Supplementing, *1-8*

Tactic, *6-3*, *6-12*, **6-12**, *6-14*, *6-30*
Atomic, **6-12**
DED, **6-13**
Default, *6-13*, **6-30**
FEQL, *6-13*, **6-14**, *6-30*
Forward, **6-14**, *6-14, 6-15*, *6-30*
GEN, **6-13**
HYP, **6-13**
Normal, **6-14**, *6-15*, *6-30*
Special, **6-13**
SUB, **6-13**
User, *6-13*, *6-21*, **6-30**
Team Library, *1-25*, *3-34*, *4-4*, *4-14*, *5-4*, *8-4*, **10-23**, *10-27*
Theory, *6-3*, **6-11**
Theory Name, *6-12*, **6-12**
TLIB, *3-37*
Tool Bar, *1-5*, *1-10*, **1-13**, *1-27*, *4-3*, *4-8, 4-9*, *4-11, 4-12*, *4-14*, *4-16*, *5-13*, *7-11*, *9-8*, *10-25*, *10-27*, *11-3, 11-4*
Top Bar, *1-3*, *1-5*, *1-10*, **1-11**, *1-14*, *1-16*, *1-27*, *4-6*, *11-5*
Translator, *1-8, 1-9*, *1-12*, *1-16*, *1-25*, *1-27*, **8-3**, *8-6*, *10-23*
TRUE, *10-21*
Type determining predicateicate, **3-3**

©1996 *B-Core (UK) Ltd.*

Type File, *4-4*, **5-3**, *7-9*
TypeChecker, **5-4**

UpperCase Identifier clause, *3-7*, *3-10*, *3-12*, *3-15, 3-16*
Users Theory
  FwdUsersTheory, **6-30**
  in Animator, **5-10**, *6-3*, *6-17, 6-18*
  in InterProver, **6-28**
  InterProver, *6-3*
  UsersTheory, **6-30**
USES Clause
  Syntax, **3-8**
  Using, *3-30*, **3-33**, *3-37*, *4-13*, *4-16*, *5-4*, *9-4*
USETAC Clause, *6-13*, **6-28**
Utilities, **1-13**, *9-9*, *10-27*, *11-3*, *11-6*

VAL BTS, *10-22*
Variables, *3-18*, **3-28**
VARIABLES Clause, **3-10**, *4-14, 4-15*, *9-4*
VMS BTS, *10-22*

 ©1996 *B-Core (UK) Ltd.*